# FlexDPDP: FlexList-based Optimized Dynamic Provable Data Possession

Ertem Esiner, Adilet Kachkeev, Samuel Braunfeld, Alptekin Küpçü, and Öznur Özkasap

Koç University, Computer Science and Engineering, İstanbul, Turkey

**Abstract**

With increasing popularity of cloud storage, efficiently proving the integrity of data stored at an untrusted server has become significant. Authenticated Skip Lists and Rank-based Authenticated Skip Lists (RBASL) have been used to provide support for provable data update operations in cloud storage. However, in a dynamic file scenario, an RBASL falls short when updates are not proportional to a fixed block size; such an update to the file, even if small, may result in $O(n)$ block updates to the RBASL, for a file with $n$ blocks.

To overcome this problem, we introduce FlexList: Flexible Length-Based Authenticated Skip List. FlexList translates variable-size updates to $O(\lceil \frac{u}{B} \rceil)$ insertions, removals, or modifications, where $u$ is the size of the update, and $B$ is the (average) block size. We further present various optimizations on the four types of skip lists (regular, authenticated, rank-based authenticated, and FlexList). We build such a structure in **O(n)** time, and parallelize this operation for the first time. We compute one single proof to answer multiple (non-)membership queries and obtain efficiency gains of **35%**, **35%** and **40%** in terms of proof time, energy, and size, respectively. We propose a method of handling multiple updates at once, achieving efficiency gains of up to **60%** at the server side and **90%** at the client side. We also deployed our implementation of FlexDPDP (DPDP with FlexList instead of RBASL) on the PlanetLab, demonstrating that FlexDPDP performs *comparable to* the most efficient *static* storage scheme (PDP), while providing dynamic data support.

**Keywords:** Cloud storage, skip list, authenticated dictionary, provable data possession, data integrity.

## 1 Introduction

Data outsourcing has become quite popular in recent years both in the industry (e.g., Amazon S3, Dropbox, Google Drive, Apple iCloud, Microsoft OneDrive) and academia [4, 6, 18, 24, 25, 37, 51, 18, 28, 52, 19, 53]. In a cloud storage system, there are two main parties, namely a server and a client, where the client transmits her files to the cloud storage server and the server stores the files on behalf of the client. For the client to be able to trust the service provider, she should be able to verify the integrity of the data. A trustworthy brand is not sufficient for the client, since hardware/software failures or malicious third parties may also cause data loss or corruption [16]. The client should be able to efficiently and securely check the integrity of her data without downloading the entire data from the server, since that will be prohibitively costly [4].

One such model proposed by Ateniese et al. is ***Provable Data Possession*** (PDP) [4] for provable data integrity. In this model, the client can challenge the server on random blocks and verify the data integrity through a proof sent by the server. PDP and related static (append-only) schemes [4, 5, 24, 37, 51] show poor performance for block-wise update operations (insertion, removal, modification). While the static scenario can be applicable to some systems (e.g., archival storage at the libraries), for several applications it is important to take into consideration the dynamic scenario, where the client keeps interacting with the outsourced data in a read/write manner, while maintaining the data possession guarantees. Scalable PDP proposed in [6] overcomes this problem with some limitations (only a pre-determined number of operations are possible within a limited set of operations). A solution called

***Dynamic Provable Data Possession*** (DPDP) proposed in [25] extends the PDP model and provides a dynamic storage scheme. Implementation of the DPDP scheme requires an underlying authenticated data structure based on a skip list [49].

A **skip list** [49] is a tree-like hierarchical key-value store whose nodes are sorted according to their keys. Authenticated skip lists are presented by [32], where skip lists and commutative hashing are employed in a data structure for authenticated dictionaries. In an authenticated skip list, each node stores a hash value calculated with the use of its associated value and the hash values of its neighboring nodes. The hash value of the root is the authentication information (meta data) that the client stores in order to verify responses from the server. To insert a new block into an authenticated skip list, one must decide on a *key* for insertion, since the skip list is sorted according to the key values. This is very useful if one, for example, inserts files into directories, since each file will have a unique name within the directory, and searching by this key is enough for general directory-related tasks. However, when one considers blocks of a file to be inserted into an authenticated skip list, the blocks do not have unique names; they have *indices*. Unfortunately, in a dynamic scenario, an insertion/deletion would necessitate incrementing/decrementing the keys of all the blocks to the right of the affected block till the end of the file, resulting in unreasonable performance.

DPDP [25] employs **Rank-based Authenticated Skip List** (RBASL) to overcome this limitation. Instead of providing an insertion *key*, the *index* where the new block should be inserted is given. These indices are imaginary (no node stores the index information). The index information can be derived using the stored *rank* information, but ranks can easily be updated such that a block insertion/deletion does *not* propagate to other blocks.

Theoretically, **an RBASL provides dynamic updates with O($\log n$) complexity**, for a file with $n$ blocks, **assuming the updates are multiples of the fixed block size**. Unfortunately, a *variable size update* leads to the propagation of changes to other blocks, making RBASL inefficient in practice. Therefore, one variable size update may affect $O(n)$ other blocks. We discuss this in Section 4.

In this paper, we propose FlexList to overcome the problem in DPDP. In FlexList, rather than *ranks*, which are indices of blocks, *lengths*, which are indices of *bytes* of data, are employed. This enables searching, inserting, removing, modifying, or challenging a specific block containing the byte at a specific index of data. Since in practice a data alteration occurs starting from a byte-index of the file, not necessarily an index of a block of the file, our DPDP with FlexList (that we call FlexDPDP) performs much faster than the original DPDP with RBASL. Even though Erway et al. [25] were the first to (informally) present the idea where the client makes updates on a range of bytes instead of blocks, we show that a naive implementation of the idea leads to insecurity in the storage system, as we show via an attack in Section 6.

We also present many **optimizations** on FlexDPDP that are observed and proposed for the first time for dynamic secure cloud storage. Our optimizations result in a dynamic cloud storage system whose efficiency is comparable to the best known static systems (PDP [4]), while still providing provable integrity for dynamically updatable data.

**Our contributions** are as follows:

- We provide the first implementation of rank-based authenticated skip list and flexible length-based authenticated skip list with the **optimal** number of links and nodes.
- We created optimized algorithms for basic operations (i.e., insertion, deletion). These optimizations are applicable to all skip list types (skip list, authenticated skip list, rank-based authenticated skip list, and FlexList).
- Our FlexList translates a *variable-sized update* to $O(\lceil \frac{u}{B} \rceil)$ insertions, removals, or modifications, where $u$ is the size of the update, and $B$ is the (average) block size (e.g., 2KB), while an RBASL requires $O(n)$ block updates in the worst case, where $n$ is the total number of blocks in the file.
- We provide, for the first time, **multi-prove** and **multi-verify** capabilities in cases where the client challenges the server for multiple blocks using authenticated skip lists, rank-based authenticated

skip lists and FlexLists. Our algorithms provide an **optimal** proof, without any repeated items. The experimental results show efficiency gains of **35%**, **35%**, **40%** in terms of proof time, energy, and size, respectively.

- We provide a novel algorithm to **build** a FlexList from scratch in $\mathbf{O(n)}$ time instead of $O(n \log n)$ (time for $n$ insertions). Our algorithm assumes the original data is already sorted, which is the case when a FlexList is constructed on top of a file in secure cloud storage. Interestingly enough, our algorithm turns out to be well parallelizable even though FlexList is an *authenticated* data structure where hashes generate *dependencies* among the nodes. Our **parallel build** algorithm achieves **speed ups of 6 and 7.7, with 8 and 12 cores**, respectively.
- We provide a **multi-block update** algorithm that achieves **60%** efficiency gains at the *server* side, compared to updating blocks independently, when the updates are on consecutive data blocks. Similarly, we achieve up to **90%** efficiency improvements at the *client* side. Our new algorithm is applicable to not only modify, insert, and remove operations but also a **mixed series of update operations**.
- We deployed our FlexDPDP client-server implementation on the network testbed **PlanetLab** and also tested its applicability on a **real SVN deployment**. The results demonstrate that our optimizations enable **4** times faster proof generation for consecutive updates in real life scenarios, and our **FlexDPDP performs comparable to the most efficient static storage scheme** (PDP), while providing dynamic data support.

The rest of the paper is organized as follows. In Section 2, our motivation behind this study is described. Section 3 introduces definitions of skip list data structure and its variants. In Section 4, details of our FlexList proposal and optimizations are described. Then, in Section 5, details of FlexDPDP that provides FlexList-based dynamic secure cloud storage are provided. Security proof of FlexDPDP and the importance of authenticating data length values are described in Section 6. Extensive analysis results including FlexList and FlexDPDP performances, deployment on the PlanetLab distributed platform, comparison with DPDP and static cloud storage on the PlanetLab are presented in Section 7. Discussion of related work is provided in Section 8, followed by conclusions and future directions in Section 9.

# 2 Motivation

In order to provide provable data possession, we employ a rooted data structure named **FlexList**, which is an authenticated dictionary. First, the file is divided into blocks (of some average size, e.g., 2KB), and the FlexList is constructed over these blocks. Each node of FlexList keeps a hash value that depends on its children, as well as the associated *length* value (similar to a *rank* in an RBASL). Thus, the rooted data structure creates dependency over all blocks (similar to a Merkle tree [42]). As in other authenticated dictionaries, any change in the FlexList nodes (e.g., their lengths), or associated blocks will result in a different root hash, due to the use of a collision-resistant hash function.

We then employ our FlexList in DPDP [25] (instead of the RBASL), and call the resulting scheme **FlexDPDP**. The meta data kept by the client is the root hash of the FlexList. The remaining parts are the same as the original DPDP construction of Erway et al. [25], though we show that the resulting efficiency gains cannot be neglected. We further optimize FlexDPDP for cloud storage operations, such as efficiently building a FlexList over a file during the initial upload, or proving and verifying *multiple* data blocks simultaneously (as suggested by PDP [4]), as well as updating multiple blocks, even with *different* update types (i.e., even for an update that involves modifications together with insertions and deletions). **With these optimizations, we show that our dynamic cloud storage scheme's efficiency is comparable to that of the best static scheme.**

**Why FlexList?** Erway et al. [25] showed that, making use of a new authenticated data structure, dynamic secure cloud storage solutions are possible. Yet, their proposal is block-wise dynamic, since they use *rank* information to organize the data items. The rank of a node in an RBASL denotes the number of blocks reachable from that node (by moving right or down, in total). As long as the updates

are of a fixed size (which is the initial block size), the scheme is efficient. By observing our own SVN repository's trace (used for over a year), we see that updates from a client are very rarely multiples of a fixed size. The SVN trace shows that the greatest common divisor of the sizes of all the commits is 1 byte. Therefore it is not possible to find a meaningful block size that is the common divisor for all updates. This shows that an update is variable sized by its nature. Therefore, a dynamic provable data possession solution should support variable block sizes.

When the updates are of variable block size, DPDP needs $O(n)$ many updates on the data structure due to the shifting of the content. We explicitly show an example in Section 4. A simple attempt to overcome this problem is padding the blocks that are shorter than the size of the fixed block size, and using regular DPDP. Unfortunately, this *hack* is not an ideal solution. Firstly, padding increases the storage requirement at the server and once in a while necessitates a reconstruction of the data structure used. The more frequent the updates are, the more reconstructions are needed. Not only that, but also current cloud services where a client wants to update their data frequently require the option for the client to update a part of their data. Thus, the client needs to access a particular part of the data. If only block-wise provable access is available due to the DPDP scheme, then for the client to find her wished part of the data becomes an operation of complexity $O(n)$, since a traverse from the start is necessary. Consider, for example, a client wishing to update her data starting from the $1879^{th}$ byte. For the server to find the desired block, he needs to go over all the blocks, and count the actual data bytes without padding (see further Section 7.4).

Moreover, since the client has no clue which index is under which block, she can be cheated by the server who may send another part of the data, claiming that it indeed contains the desired byte (see Section 6). This insecurity may be tried to be solved via also authenticating the padding lengths together with the actual data lengths, but instead we propose FlexList, which provides an elegant, provably secure, and efficient way of dealing with variable-sized blocks, without any complications or storage overhead of padding.

# 3 Definitions

**Skip List** is a probabilistic data structure presented as an alternative to balanced trees [49]. It is easy to implement without complex balancing and restructuring operations such as those in AVL or Red-Black trees [2, 8]. A skip list keeps its nodes ordered by their *key* values. We call a leaf-level node and all nodes directly above it at the same index a *tower*.

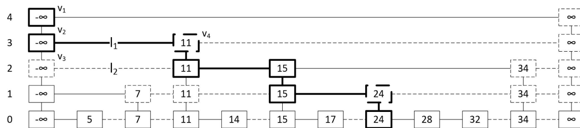


Figure 1: A regular skip list with the search path of the node with the key 24 highlighted. Numbers on the left represent levels. Numbers inside nodes are key values. Dashed lines indicate unnecessary links and nodes.

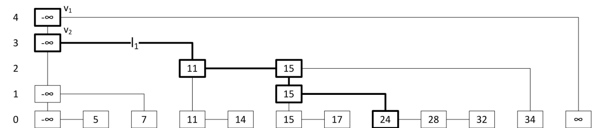


Figure 2: Skip list of Figure 1 without unnecessary links and nodes.

Figure 1 demonstrates a search on a skip list. The search path for the node with key 24 is highlighted in bold. In a basic skip list, the nodes include *key*, *level*, and data (only at the leaf level) information, and $below$ and $after$ links (e.g., $v_2.below$ = $v_3$ and $v_2.after$ = $v_4$). To perform the search for 24, we start from the root ($v_1$) and follow the $below$ link to $v_2$, because $v_1$'s $after$ link leads to a node which has a key greater than the key we are searching for ($\infty > 24$). Then, from $v_2$ we follow the link $l_1$ to $v_4$, since the key of $v_4$ is smaller than (or equal to) the searched key ($11 \leq 24$). In general, if the key of the node where the $after$ link leads is smaller or equal to the key of the searched node, we follow that link; otherwise we follow the $below$ link. Using the same decision mechanism, we follow

the highlighted links until the searched node is found at the leaf level (if it does not exist, then the node with key immediately before the searched key is returned).

We observe that some of the links are never used in the skip list, such as $l_2$, since any search operation with key greater or equal to 11 will definitely follow the link $l_1$, and a search for a smaller key would never advance through $l_2$. Thus, we say links, such as $l_2$, that are *not* present on *any* search path are *unnecessary*. When we remove the unnecessary links, we observe that the non-leaf-level nodes that are left without $after$ links (e.g., $v_3$) are also unnecessary, since they do not provide any new dependencies in the skip list. Every time they are visited, the search will just continue following the $below$ link of such a node. Although it does not change the asymptotic complexity, it is beneficial not to include them for time and space efficiency, as our experimental results confirm. An optimized version of the skip list from Figure 1 can be seen in Figure 2 with the same search path highlighted. Formally:

- A **link** is **necessary** if and only if it is on some search path.
- A **node** is **necessary** if and only if it is at the leaf level or has a necessary $after$ link.

Assuming existence of a **collision-resistant hash function** family $H$, we randomly pick a hash function $h$ from $H$ and let $||$ denote concatenation. Throughout our study we will use: $hash(x_1, x_2, ..., x_m)$ to mean $h(x_1||x_2||...||x_m)$.[1]

An **authenticated skip list** is constructed with the use of a collision-resistant hash function and keeps a hash value at each node [33]. The hash value is calculated with the following inputs: *level* and *key* of the node, and the hash values of the node $after$ and the node $below$. If there is no $after$ neighbor, then a dummy value (e.g., null) is used in the hash calculation. Through the inputs to the hash function, all nodes are dependent on their $after$ and $below$ neighbors. Thus, the root node is dependent on every node. Leaf nodes (at level 0) keep links to the file blocks (or to different structures e.g., files, directories, *tags*: anything to be kept intact). If any node of the skip list is modified, or the values linked at the leaf level nodes are modified, then this will result in a different hash of the root. Due to the collision resistance of the hash function, knowing the hash value of the root is sufficient for later integrity checking.

A **rank-based authenticated skip list (RBASL)** is different from an authenticated skip list by means of how it indexes data [25]. An RBASL has *rank* information (used in hashing, replacing the *key*), meaning how many leaf-level nodes are reachable from that node. An RBASL is capable of performing all operations that an authenticated skip list can, in the cloud storage context.

# 4 FlexList

FlexList is an **authenticated** data structure (an example is shown in Figure 3), where each node keeps a **hash** value calculated according to its *rank*, *level*, *the hash value of below neighbor*, and *the hash value of the after neighbor*, where **rank** is the number of *bytes* that can be reached from that node and **level** is the height of a node in the FlexList. Each leaf level node keeps a link to the **data** (a block of the file stored) that it refers to, the **length** of the data, and a **tag** value calculated according to that data. **Rank** values are calculated by adding the below and after neighbors' ranks. If a node is at the leaf level, we use the *length* of its data *as the below neighbor's rank*. Leaf level nodes' hash calculation is slightly different. As *the hash value of below neighbor*, we use the associated *tag*, and in addition, the *length* value is also included in the hash calculation. Note that, the root node is dependent to all (leaf level) nodes.

FlexList has **sentinel** nodes as the first and last nodes. Sentinel nodes have no data; hence their length values are 0 and they do not affect the rank values of the other nodes. These nodes generate no new dependencies, but are useful to make algorithms easier and more understandable.

[1] We assume the necessary care is taken to fix the bitlengths of the concatenated values so that the hash is still collision resistant.

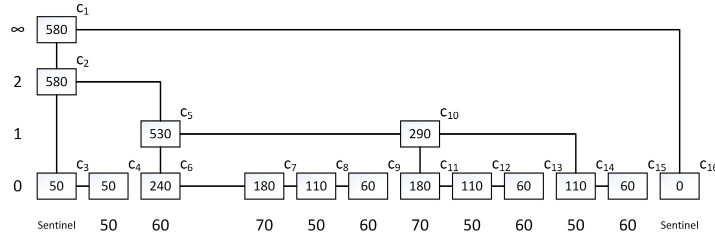


Figure 3: A FlexList example. The numbers at the bottom are the lengths of the data blocks associated with the leaf level nodes. The numbers in the nodes are their ranks. The numbers at the left denote levels.

A **FlexList supports variable-sized blocks**, whereas an RBASL can only be used with fixed block size (since operations by the **byte-index** of the data are not possible with the rank information of an RBASL). In FlexList, instead of using the block indices as ranks as in the RBASL, we use byte indices. Thus, the information stored at each node denotes the number of *bytes* of data reachable from that node. We show the advantage of this approach via an example.

Consider Figure 4-A that represents an outsourced file divided into blocks of fixed size. In our example, the client wants to change the file composed of the text "The quick brown fox jumps over the lazy dog..." such that "brown" becomes "red". The client's diff algorithm returns "[delete from index 11 to 15] and [insert "red" starting from index 11]". Apparently, a modification to the $3^{rd}$ block will occur. With a rank-based skip list, to continue functioning properly, a series of updates is required as shown in Figure 4-B, which asymptotically corresponds to $O(n)$ alterations (assuming no padding, see Section 7.4). Therefore, for instance, an RBASL having 500000 leaf-level nodes needs an expected 250000 update operations for a single variable-sized update, such as the one in this example. Besides the modify operations and related hash calculations, this also corresponds to 250000 new tag calculations either on the server side, where the private key (the order of the RSA group) is unknown (and thus computation is very slow), or at the client side, where all the new tags should go through the network. Furthermore, a verification process for the new blocks is also required (which means a huge proof sent by the server and verified by the client, who needs to compute an expected 375000 hash values). With our proposed FlexList, only one modification suffices, as indicated in Figure 4-C.

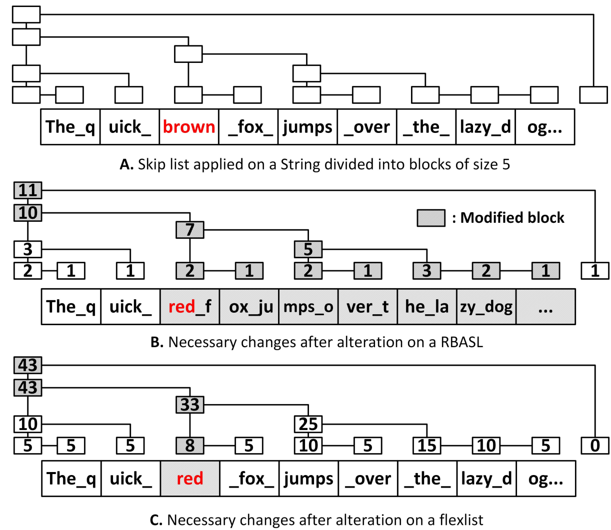


Figure 4: Skip list alterations depending on an update request.

FlexList overcomes the problem of RBASL, which is not capable of providing variable block sized operations. At each node, a FlexList stores the number of *bytes* that can be reached from that node, instead of number of *blocks* that are reachable. The *rank* of each leaf-level node is computed as the sum of the *length* of its data and the *rank* of the $after$ node (0 if *null*). The *length* information of each data block is added as a parameter to the hash calculation of that particular block. We discuss the insecurity of an implementation that does not include the length information in the hash function calculation in Section 6. Note that when the length of the data at each leaf is considered as a unit, the FlexList reduces to an RBASL (thus, ranks only count the number of reachable blocks). Therefore all our optimizations are also applicable to RBASL, which is a special case of FlexList.

### 4.1 Preliminaries

In this section, we introduce the helper methods required to traverse the skip list, create missing nodes, delete unnecessary nodes, delete nodes, and decide on the level to insert at, to be used in the essential algorithms (*search, modify, insert, remove*).

To make our FlexList algorithms easier to understand, we define the **sub skip list** concept. An example is illustrated in Figure 5. Let the search index be 250 and the current node start at the root ($v_1$). The search is similar to an RBASL. At each node, we compare the rank of the $below$ node to see if we can reach the searched index via that node. In this example, the $below$ node of $v_1$ is $v_2$, who can reach 350 bytes of data. Since we are searching for the index 250, we continue with $v_2$. The current node follows its $below$ link to $v_2$ and enters a sub skip list (big dashed rectangle).

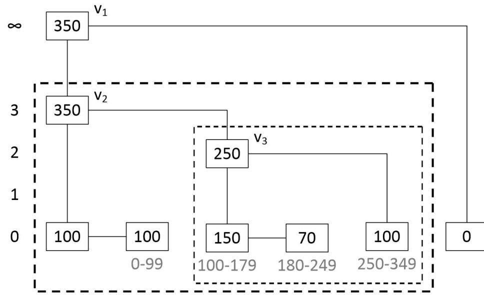


Figure 5: A FlexList example with 2 sub skip lists indicated.

Now, $v_2$ is the root of this sub skip list and the searched node is still at index 250. We again look at the $below$ node of $v_2$. But this time it is useless for our purposes, since it can only reach 100 bytes whereas we are searching for 250. Thus, we should continue with the $after$ node. In order to reach the searched node, the current node moves to $v_3$, which is the root of another sub skip list (small dashed rectangle).

Note that by moving $after$, we have skipped over some data equivalent to the rank of the $below$ node we skipped over. We were originally searching for index 250, but now that we have skipped over 100 bytes of data, in this sub skip list, we continue our search with $250 - 100 = 150$. The amount reduced from the search index is equal to the difference between the rank values of $v_2$ and $v_3$, which is equal to the rank of $below$ of $v_2$. Whenever the current node follows an $after$ link, the search index should be updated, since some data is skipped over. On the other hand, if a $below$ link is followed, no data is skipped, and thus the search index stays the same. To finish the search, the current node follows the $after$ link of $v_3$ to find the node containing the searched index.

We now present our utility functions as a preliminary to understanding the main methods. Note that all algorithms are designed to fill a stack $\sqcup_n$ that stores nodes which may need a recalculation of the hash values if the data structure authenticated, and rank values if used in a FlexList or RBASL. All algorithms that move the current node $cn$ immediately push the new current node to the stack $\sqcup_n$ as well. All these algorithms are provided in the Appendix, together with Table 4, which shows the notation used.

**canGoBelow** (Algorithm A.1) and **canGoAfter** (Algorithm A.2) are the two methods that constitute the decision mechanism of the traversal in the FlexList. They check if the searched index can be found following the $below$ or $after$ link, respectively. There are some slight differences depending on from which other algorithm these are called. They are stated as comments in the algorithms.

**nextPos** (Algorithm A.3): The $nextPos$ method moves the current node $cn$ repetitively until the position desired by the calling method ($search$, $insert$, $remove$) is reached. There are 4 cases for $nextPos$ depending on the caller:

- $insert$ - moves the current node $cn$ to the node immediately before the insertion point at the insertion level.
- loop in $insert$ - moves $cn$ until it finds the next insertion point for a new node.
- $remove$ or $search$ - moves the current node $cn$ until it finds the searched node's *tower*.
- loop in $remove$ - moves the current node $cn$ until it encounters the next node to delete.

**tossCoins** (no pseudocode given): Probabilistically determines the level value for a new node tower. A coin is tossed until it comes up heads. The output is the number of consecutive tails.

**createMissingNode** (Algorithm A.4) is used in both the *insert* and *remove* algorithms. Since in a

FlexList there are only necessary nodes, when a new node needs to be connected, this algorithm creates any missing node to make the connection.

**deleteUNode** (Algorithm A.5) is employed in the *remove* and *insert* algorithms to delete an *unnecessary* node (this occurs when a node loses its *after* node, except at the leaf level) and maintain the links. It takes the previous node and current node as inputs, where the current node is unnecessary and meant to be deleted. We preserve connections between the necessary nodes after the removal of the unnecessary one. This includes the deletion of the current node if it is not at the leaf level. It sets the previous node's $after$ or $below$ to the current node's $below$. As the last operation of deletion, we remove the top node from the stack $\sqcup_n$, as its rank and hash values no longer need to be updated.

**deleteNode** method, employed in the *remove* algorithm, takes two consecutive nodes, the previous node and the current node. By setting $after$ pointer of the previous node to current node's $after$, it detaches the current node from the FlexList.

### 4.2 Methods of FlexList

We first describe the basic functions of FlexList such as search, modify, insert, and remove. Then, we show how to employ these functions in a secure cloud storage system. All algorithms are designed to fill a stack for the possibly affected nodes, whose rank and hash values may need re-calculation. The nodes in the stack constitute a *search path*, which is the basis of a *proof path*.

**search** (Algorithm A.6) is used to find a particular index. It takes the index $i$ as the input, and outputs the node at index $i$ and the stack $\sqcup_n$ filled with the nodes on the search path. Any value between 0 and the file size in bytes is valid to be searched. It is not possible for a valid index not to be found in a FlexList.

In the search algorithm, the current node $cn$ starts at the root. The $nextPos$ method moves $cn$ to the position just before the top of the tower of the searched index. Then $cn$ is taken to the searched node's tower and moved all the way down to the leaf level. The leaf-level node at the searched index is returned. All the nodes along the path from the root to this node are added to the stack that is returned as well.

**modify** (Algorithm A.7): By taking index $i$ and some new data as input, this algorithm first uses the *search* algorithm to find the node that includes the data at index $i$, and then updates its data. The outputs are the modified node and the stack $\sqcup_n$ filled with nodes on the search path. Afterwards, this stack is used to recalculate the hash values of the nodes along the search path, in a bottom-up fashion.

**insert** (Algorithm A.8) is run to add a new node to the FlexList by adding new nodes along the insertion path. The inputs are the index $i$ and $data$. The algorithm generates a random insertion *level* by tossing coins, then creates the new node with the given $data$ and attaches it to the index $i$, along with the necessary nodes until the *level*. Note that this index should be the beginning index of an existing node, since inserting a new block inside an existing block makes no sense.[2] As output, the algorithm returns the stack $\sqcup_n$ filled with nodes on the search path of the new block.

Figure 6 demonstrates the insertion of a new node at index $450$ with $level$ $4$. $nextPos$ brings the current node to the closest node to the insertion point with level greater than or equal to the insertion level ($c_1$ in the figure). We create any missing node at the $level$, if there was no node to connect the new node to (in our example, $m_1$ is created to connect $n_1$ to). Within the while loop, during the first iteration, $nextPos$ adds $c_2$, $c_3$, and $d_1$ to the stack, and stops at $d_1$. Then a new node $n_1$ is created and inserted to level 2, after $m_1$. Inserting $n_1$ makes $d_1$ unnecessary, since $n_1$ stole its after link (i.e., $n_1.after = v_4$ now, and $d_1.after$ is $null$). Thus, $d_1$ is deleted by $deleteUNode$, and removed from the stack. Likewise, in the next iteration $n_2$ is created and inserted at level $1$ (as below of $n_1$). Since $n_2$ steals the after link of $d_2$ (i.e., $n_2.after = v_3$ now, and $d_2.after$ is $null$), $d_2$ is removed as unnecessary. Note

[2] In case of an addition inside a block, we can do the following: search for the block including the byte where the insertion will take place, add our data in between the first and second part of data found to obtain new data and employ *modify* algorithm (if new data is long, we can divide it into parts and send it as one modification and a series of insertions).

that removal of $d_1$ and $d_2$ results in $c_3$ getting connected to $v_1$ (i.e., first, when $d_1$ is removed, $c_3.after$ becomes $d_2$. Then, when $d_2$ is removed, $c_3.after$ becomes $v_1$.). The last iteration inserts $n_3$, and places data. Since this is a FlexList, hashes and ranks of all the nodes in the stack will be recalculated (i.e., stack contains $c_1, m_1, n_1, c_2, c_3, n_2, n_3, v_1, v_2$). Those are the only nodes whose hash and rank values might have changed.

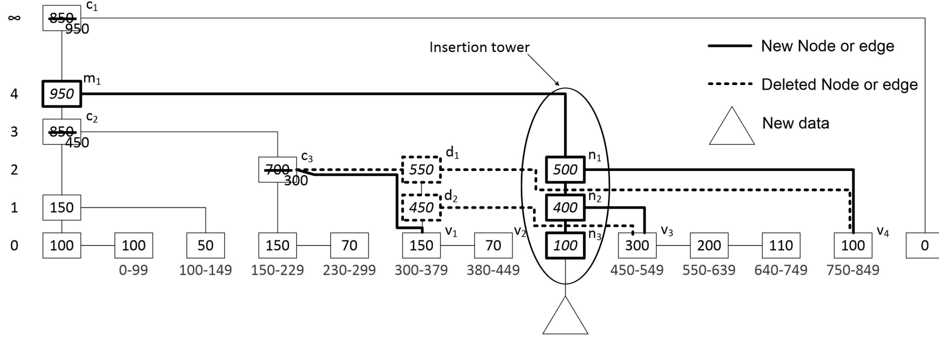


Figure 6: Insert at index 450, level 4 (FlexList).

**remove** (Algorithm A.9) is run to remove the node that starts with the byte at the index $i$. As input, it takes the index $i$. The algorithm detaches the node to be removed and all other nodes above it (i.e., the tower) while preserving connections between the remaining nodes. The algorithm returns the stack $\sqcup_n$ filled with the nodes on *the search path of the left leaf-level neighbor of the node removed.*

Figure 7 demonstrates removal of the node having the byte with index 450. The algorithm starts at the root $c_1$, and stops first at $d_1$, since that node points to our deletion tower. The tower is to be removed, and hence $d_1$ is no longer necessary. Therefore, $d_1$ is deleted, $c_1.below$ becomes $c_2$, and the search continues. Next, we go find the next node pointing to the deletion tower. This node is $c_3$, which is currently connected to $d_2$. Since $d_2$ is to be deleted, creating of a missing necessary node $m_1$ is required such that $m_1.after = v_4$. Once $m_1$ is created at the same level as $d_2$ and $d_2$ is deleted, $c_3.after$ becomes $m_1$. Continuing our search, we now stop at $m_1$, delete $d_3$, create a missing node $m_2$ at the same level as $d_3$, such that $m_2.after = v_3$. The last iteration moves the current node to $v_2$ and deletes $d_4$ without creating any new nodes, since we are at the leaf level. The output stack contains nodes $(c_1, c_2, c_3, m_1, m_2, v_1, v_2)$. Rank and hash values of those nodes could have changed, those values will be recalculated.

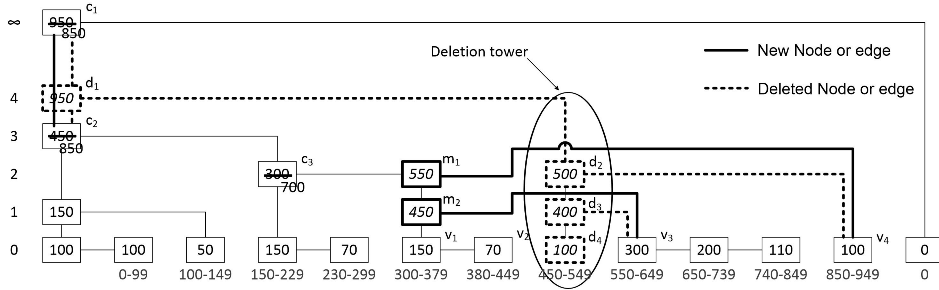


Figure 7: Remove block at index 450 (FlexList).

## 4.3 Novel Build from Scratch Algorithm

The usual way to build a skip list (or FlexList) is to perform $n$ insertions (one for each underlying item). Such an approach will result in $O(n \log n)$ total time complexity. But, when the underlying data is already sorted, as in the secure cloud storage scenario (where the blocks of a file are already sorted by their indices), a much more efficient algorithm can be developed.

Observe that a skip list contains $2n$ nodes in expectation [49]. The $O(n \log n)$ complexity is a result of the sorting requirement. Therefore, on already sorted data, we present our novel algorithm for

building a FlexList from scratch in $O(n)$ time. To the best of our knowledge, such an efficient build algorithm was not proposed before, not even for regular skip lists.

**buildFlexList** (Algorithm A.10) algorithm generates a FlexList over a set of sorted data in time complexity $O(n)$ with high probability. It has the small space complexity of $O(l)$ where $l$ is number of levels in the FlexList (i.e., $l = O(\log n)$ with high probability). As the inputs, the algorithm takes the list of blocks $B$ on which the FlexList will be generated, the corresponding (randomly generated) levels $L$ and tags $T$. The algorithm assumes data is already sorted. In cloud storage, the blocks of a file are already sorted according to their block indices, and thus our optimized algorithm perfectly fits our target scenario. As the output, the algorithm returns the root node of the constructed FlexList. It works from right to left and bottom to top, thus performing a single pass over the nodes.

Figure 8 demonstrates an example FlexList build process where the insertion levels of blocks are 4, 0, 2, 0, 0, 0, 1, 0, 0, 1, 0, in order. Labels $v_i$ on the nodes indicate the generation order of the nodes. The idea of the algorithm is to build towers for each block, up to the associated level. As shown in the figure, all towers have only one link from left side to its tower head (the highest node in the tower). For example, the head of the tower composed of $v_6, v_7$ is $v_7$ , and there is only one link coming to that tower from left, which is the link from $v_{12}$ to the tower head $v_7$. Therefore, we need to store the tower heads in a vector, and then make necessary connections.

At a high level, the algorithm starts creating the rightmost tower first. The algorithm then remembers its tower head at the associated level. Then, the second rightmost tower is created. If there are tower heads to the right of it that it should be connected to, those connections are created. For example, when it is the turn of the $v_6, v_7$ tower, we know that it should be connected to $v_5$ at level 0, $v_3$ at level 1. Once the leftmost tower is created and connected to other towers, the algorithm returns the root. During this process, the ranks and hashes of each new node are also calculated. Thus, the build algorithm not only can work for a regular skip list, but can also work for authenticated and rank-based versions, such as FlexList or RBASL, as well. Note that this build algorithm creates an optimal skip list without unnecessary nodes and links, as $n$ insertions with our optimal insertion algorithm would have built. The full algorithm, as well as its detailed explanation using Figure 8, is provided in the Appendix.

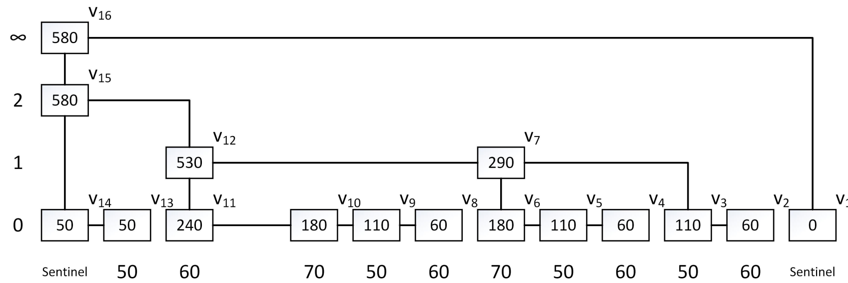


Figure 8: buildFlexList example.

## 4.4 Parallel Build FlexList

However unintuitive, we can also parallelize the build process of even an *authenticated* data structure, such as FlexList. Figure 9 shows the three-core parallel construction of the FlexList (example in Figure 8). The approach consists of three steps: First, the tasks are distributed to threads and FlexLists for subsets of blocks are generated. Second, to unify them into a single FlexList, all the roots are connected together with links ($c_1$ is connected to $r_1$, eliminating $l_1$, and $r_1$ is connected to $r_2$, eliminating $l_2$) and new rank values of the modified roots ($r_1$ and $c_1$) are calculated. Third, FlexList remove function is used to remove the unnecessary sentinels directly below the roots of the sub-FlexLists that remain in between each part (remove indices $360 = c_1.rank - r_2.rank$ and $180 = c_1.rank - r_1.rank$). In the example, the remove operation generates $v_{12}$ and $v_7$ of Figure 8 and connects the remaining nodes to them, and rank values on the search paths of $c_2$, $c_6$, $c_7$, $c_{11}$ are recalculated after the removal of the unnecessary sentinel

nodes. As a result, all the nodes of the sub-FlexLists are connected to their levels on the FlexList, and the same FlexList of Figure 8 is obtained efficiently in a parallel manner.

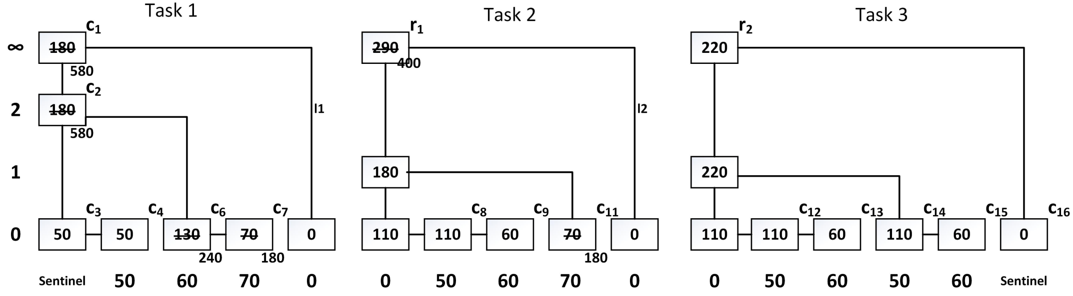

Figure 9: A build skip list distributed to 3 cores.

# 5 FlexList-based Dynamic Secure Cloud Storage

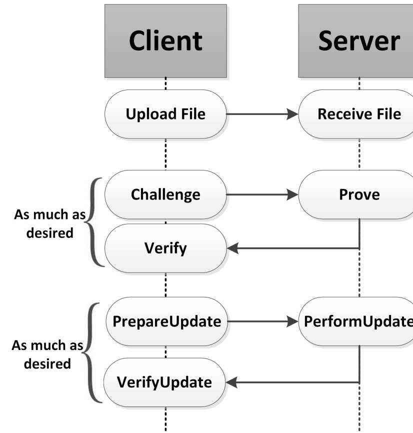


Figure 10: Client Server interactions in FlexDPDP.

In this section, we describe the application of FlexList to integrity check in secure cloud storage systems according to the DPDP model [25]. We call the resulting scheme **FlexDPDP** that includes the new capabilities and efficiency gains discussed in Section 4. The FlexDPDP uses *homomorphic verifiable tags* [5, 4, 25]: multiple tags can be combined to obtain a single tag that corresponds to a single combined block. Tags are small compared to data blocks, enabling storage in main memory. Authenticity of the skip list guarantees integrity of tags, and tags protect the integrity of the data blocks.

The DPDP model has two main parties: the client and the server. The cloud server stores a file on behalf of the client. It has been shown that an RBASL can be created on the outsourced file to provide proofs of integrity as depicted in Figure 10. The following algorithms are used:

- $Challenge$ is run by the client to request a proof of integrity for randomly selected blocks.
- $Prove$ is run by the server in response to a challenge, to generate the proof of possession.
- $Verify$ is run by the client upon receipt of the proof. A return value of accept means the file is kept intact by the server.
- $prepareUpdate$ is run by the client when she changes some part of her data. The client sends the update information to the server.
- $performUpdate$ is run by the server in response to an update request to perform the update *and* prove that the update is performed reliably.
- $verifyUpdate$ is run by the client upon receipt of the proof of the update, and it returns accept (and updates her meta data) if the update was performed reliably.

### 5.1 Preliminaries

Before providing optimized proof generation and verification algorithms, we introduce essential methods used in our algorithms to determine intersection nodes, search multiple nodes, and update rank values. Table 5 in the Appendix shows the notation used in this section.

**Proof node** is the building block of a proof. It contains level, data length (if level is 0), rank, hash, and three boolean values ($rgtOrDwn$, end flag, and intersection flag). Level and rank values belong to the actual node for which the proof node is generated. The hash is the hash value of the neighbor node that is not on the search path. There are two scenarios for setting hash and $rgtOrDwn$ values:

1. When the search path follows the $below$ link of the node in consideration, the hash of the corresponding proof node is set to the hash of the current node's $after$ (since that part is not on the search path) and its $rgtOrDwn$ value is set to $dwn$ (since the search path continues down).
2. When the search path follows the $after$ link of the node in consideration, the hash of the corresponding proof node is set to the hash of the current node's $below$ (since that part is not on the search path) and its $rgtOrDwn$ value is set to $rgt$ (since the search path continues right).

**isIntersection**: This function is used in *searchMulti* (defined later) for checking whether or not a given node is an intersection. It takes current node, the challenged indices, two indices (which will be checked for intersection, in ascending order), and the rank state (to know which sub skip list is considered). A node is an *intersection* point of proof paths of two searched indices when the first index can be found following the $below$ link and the second index is found by following the $after$ link of that node. The *isIntersection* method not only returns true or false according to this criteria, but also helps the proof generation by finding the next intersection in the challenge vector $C$. In practice, among the two indices given to this function, the first one will be the index we are currently searching for (to create a proof), and the second one will be the index of some other challenged index. Note that this method directly returns false if there is only one challenged index, or if the first and last indices are not intersecting at the given node.

**searchMulti** (Algorithm A.11): This algorithm is used in the *genMultiProof* algorithm (to generate a combined proof for multiple challenged indices, see Section 5.2) to generate the proof path for multiple nodes without unnecessary repetitions of the proof nodes. The idea is to search for the indices, create a proof node for every node on the search path, and note down any intersections. Figure 11, where the node at the index $450$ is challenged, demonstrates an example of how the algorithm works. The aim is to provide the proof path for the challenged node. We assume that in the search, the current node $cn$ starts at the root ($w_1$ in the example). Therefore, initially the search index $i$ is $450$, the proof vector $P$ and intersection stack $\sqcup_s$ are empty.

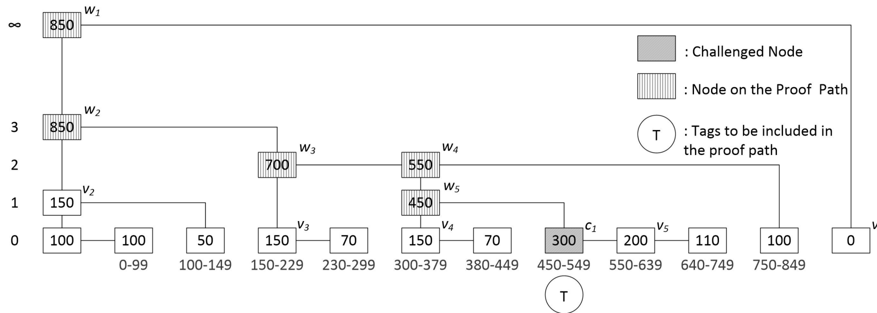


Figure 11: Proof path for challenged index 450 in a FlexList.

For $w_1$, a proof node $p$ is generated using scenario (1) (see proof node description), where $p$.hash is set to $v_1$.hash and $p.rgtOrDwn$ is set to $dwn$. For $w_2$, the proof node $p$ is created as described in scenario (2) above, where $p$.hash is set to $v_2$.hash and $p.rgtOrDwn$ is set to $rgt$. The proof node for $w_3$ is created using scenario (2). For $w_4$ and $w_5$, proof nodes are generated as in scenario (1). The last

node $c_1$ is the challenged leaf node, and the proof node for this node is also created as in scenario (1). Note that in the second, third, and fifth iterations of the while loop, the current node is moved to a sub skip list. Setting the end flag and collecting the data length, as well as setting the intersection flag and saving the rank state are crucial for generation of proof for multiple blocks. The full algorithm is in the Appendix.

**updateRankSum**: This algorithm, used in the *verifyMultiProof* algorithm (to verify the proof of multiple challenged indices, see Section 5.2), is given the rank difference as input, the verify challenge vector $V$, and indices $start$ and $end$ (on $V$). The output is a modified version $V'$ of the verify challenge vector. The procedure is called when there is a transition from one sub skip list to another (larger one, since verification is bottom-up and right-to-left). The method updates entries starting from index $start$ to index $end$ by rank difference, where rank difference is the size of the larger sub skip list minus the size of the smaller sub skip list.

**Tag calculation and the Block Sum**: We use an RSA group $Z^*_N$, where N = $pq$ is the product of two large prime numbers, and $g$ is a high-order element in $Z^*_N$. It is important that the server does not know $p$ and $q$. The *tag* $t$ of a block $m$ is computed as $t = g^m \mod N$. The *block sum* is computed as $M = \sum_{i=0}^{|C|} a_i m_{C_i}$ where $C$ is the challenge vector containing block indices and $a_i$ is the random value associated with the $i^{th}$ challenge index. These are directly taken from the original DPDP [25].

## 5.2 Handling Multiple Challenges at Once

Client-server interaction in secure cloud storage (Figure 10) starts with the client pre-processing her data (creating a FlexList for the file and calculating tags for each block of the file). The client sends the random seed she used for generating the FlexList to the server along with the data, the tags, and the public key $(g, N)$ used for computing the tags. Using the seed, the server constructs a FlexList over the blocks of data and assigns tags to leaf-level nodes. Note that the client may request the root value calculated by the server to verify that the server constructed the correct FlexList over the file. When the client checks and verifies that the hash of the root value is the same as the one she had calculated, she may safely remove her data and the FlexList. The client keeps the hash of the root of the FlexList as meta data for later use in the proof verification mechanism.

To challenge the server, the client generates two pseudo-random generator seeds, one that will generate the challenge vector $C$ of random indices for bytes to be challenged, and another that will generate random coefficients $a_i$ to be used in the block sum [4]. The client sends these two seeds to the server as the challenge, and keeps them for verification of the server's response.

### 5.2.1 Proof Generation

**genMultiProof** (Algorithm A.12): Upon receipt of the random seeds from the client, the server generates the challenge vector $C$ and random values $a_i$ according to the pseudo-random generators, and runs the *genMultiProof* algorithm in order to get tags, file blocks, and the proof path for the challenged indices. The algorithm searches for the leaf node of each challenged index and stores all nodes across the search path in the proof vector. However, when multiple blocks are challenged (as in PDP [4] and DPDP [25]), if we start from the root for each challenged block, there will be a lot of replicated proof nodes along these multiple search paths. In the example of Figure 12, if the proofs were generated individually for each challenged index, $w_1$, $w_2$, and $w_3$ would be replicated 4 times (because they are common to the proof path of each challenged index), $w_4$ and $w_5$ would be replicated 3 times (common for all challenged ones except $c_1$), and $c_3$ would be replicated 2 times. To overcome this problem, we save the state at each intersection node. For example, in Figure 12, the algorithm starts traversing from the root ($w_1$), searching for the leftmost challenged node ($c_1$). Next, the algorithm continues searching for the next challenged node ($c_2$), but starting from $w_3$ (which is the intersection node for $c_1$ and $c_2$)

instead of $w_1$. The proof generation for $c_3$ will continue from $w_5$ (the intersection node for $c_2$ and $c_3$), and so on.

In our **optimal** proof, only one proof node is generated for each node on *any* proof path. This is beneficial in terms of not only space but also time (see Section 7), and hence is much better than first generating individual proofs and then performing a replica-removing merge or applying compression. Correspondingly, the verification time of the client is also greatly reduced since she computes fewer hash values. The full algorithm and its detailed explanation using Figure 12 are in the Appendix. At the end of the *genMultiProof* algorithm, the proof vector (Figure 13), the tag vector, and the block sum are sent to the client for verification.

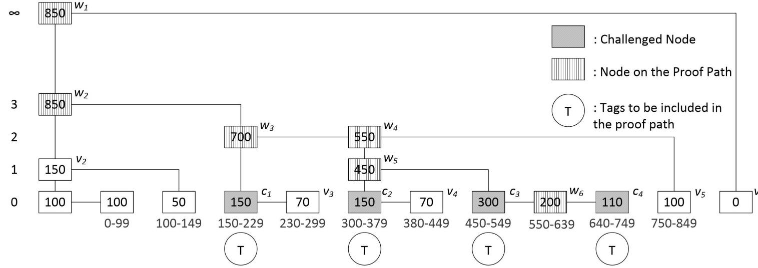


Figure 12: Multiple blocks are challenged in a FlexList.

| | |
|---|---|
| $c_4$ | 0, 110, - , dwn, E, 110 |
| $w_6$ | 0, 200, $w_6$.Tag, rgt |
| $c_3$ | 0, 300, -, dwn, I, E, 100 |
| $c_2$ | 0, 150, $v_4$.hash, dwn, E, 80 |
| $w_5$ | 1, 450, -, dwn, I |
| $w_4$ | 2, 550, $v_5$.hash, dwn |
| $c_1$ | 0, 150, $v_3$.hash, dwn, E, 80 |
| $w_3$ | 2, 700, -, dwn, I |
| $w_2$ | 3, 850, $v_2$.hash, rgt |
| $w_1$ | ∞, 850, $v_1$.hash, dwn |

A proof Node (a tuple in proof vector) contains the following:
- Level
- Rank
- Hash of neighbor not included in the proof vector
- Direction of the next proof node relative to this one
- Intersection flag
- End flag
- Data length

Figure 13: Proof vector for Figure 12 example. "I" denotes intersection, "E" denotes end, and data length is included only for end nodes.

#### 5.2.2 Verification

**verifyMultiProof** (Algorithm A.13): Recall that the client keeps random seeds used for the challenge, and generates the challenge vector $C$ and random values $a_i$ according to these seeds. If the server was honest, these will contain the same values as the ones the server generated for preparing the proofs. There are two steps in the verification process: tag verification and FlexList verification.

**Tag verification** is done as follows: Upon receipt of the tag vector $T$ and the block sum $M$, the client calculates $tag = \prod_{i=0}^{|C|} T_i^{a_i} \mod N$ and accepts iff $tag = g^M \mod N$. This part proves that the file blocks match the tags sent (as in PDP [4] and DPDP [25]). Next, FlexList verification checks that the tags remain intact. Thus, together they prove file integrity.

**FlexList verification** involves calculation of hashes for the proof vector $P$ (Figure 13) treated as a stack. The calculation of hashes is done in the reverse order of the proof generation in *genMultiProof* algorithm. Therefore, the calculations are performed in the following order: $c_4$, $w_6$, $c_3$, $c_2$, $w_5$, ... until the hash value for the root (the last element in the stack) is computed. Observe that to compute the hash value for $w_5$, for example, the hash values for $c_3$ and $c_2$ are needed, and this reverse (top-down) ordering always satisfies these dependencies. Note also that this top-down ordering on the stack corresponds to a bottom-up and right-to-left traverse in a FlexList, and as such, the hash calculations respect all dependencies, and no secondary pass over the proof vector is necessary. When the hash for the last proof node of the proof path is calculated ($w_1$), it is compared with the meta data that the client possesses. Detailed explanation is in the Appendix.

This check makes sure that the nodes, whose proofs were sent, are indeed in the FlexList that corresponds to the meta data stored at the client. But the client also has to make sure that the server indeed proved storage of the data that she challenged. The server may have lost those blocks but may instead be proving storage of some other blocks at different indices. To prevent this, the verify challenge

vector, which contains the start indices of the challenged nodes (150, 300, 450, and 460 in our example), is generated by the rank values included in the proof vector (in lines 11, 23, and 27 of Algorithm A.13). With the start indices and the lengths of the challenged nodes given, we check if each challenged index in $C$ is included in a node that the proof is generated for (as shown in line 32). For instance, we challenged index 170. From the proof, we observed that $c_1$ starts from index 150 and is of length 80. We check if $0 \leq 170 - 150 < 80$. Such a check is performed for each challenged index and each proof node with an *end* mark. The algorithm is provided in the Appendix.

## 5.3 Handling Multiple Updates at Once

### 5.3.1 Proving Variable-size Multi-block Updates

We investigated the verifiable updates and inferred that the majority of the time spent is for the hash calculations in each update. When a client alters her data and sends it to the server, she generates a vector of updates ($U$) out of a diff algorithm, which is used to show the changes between the current and the former versions of a file.

An update information $u \in U$, includes an index $i$, and (if insert or modify) a block and a tag value. Furthermore, the updates on a FlexList usually consist of a series of *modify* operations followed by either *insert* or *remove* operations, all to adjacent nodes. This nature of the update operations makes single updates inefficient since they keep calculating the hash values of the same nodes over and over again. To overcome this problem, we propose dividing the task into two: doing a series of **updates without the hash calculations**, and then calculating all **affected nodes' hash values at once**, where affected means that at least an input of the hash calculation of that node has changed. The **multiUpdate** (Algorithm A.14) gets a FlexList and a vector $U$ of updates, and produces a proof vector $P$, a tag vector $T$, a block sum $M$, and the new hash value $newRootHash$ of the root.

We illustrate handling multiple updates with an example. Consider a *multiUpdate* called on the FlexList of Figure 3 and a consecutive modify and insert happen to indices 50 and 110 respectively (insert level is 2). When the updates are done without hash calculations, the resulting FlexList looks like the one in Figure 14. Since the tag value of $c_6$ has changed and a new node added between $c_6$ and $c_7$, all the nodes getting affected should have a hash recalculation. If we first perform the insert, we need to calculate hashes of $n_3$, $n_2$, $c_6$, $n_1$, $c_2$ and $c_1$. Later, when we do the modification to $c_6$ we need to recalculate hashes of nodes $c_6$, $n_1$, $c_2$ and $c_1$. There are 6 different nodes to recalculate hashes of, but we do 10 hash calculations. Instead, we propose performing the insert and modify operations and then calculating the necessary hash values without any wasted effort. The detailed algorithms are in the Appendix.

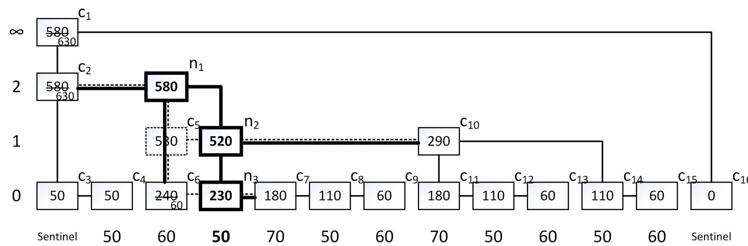


Figure 14: Insert and remove examples on FlexList.

| | |
|---|---|
| $c_4$ | 0, 110, - , dwn, E, 110 |
| $w_6$ | 0, 200, $w_6$.Tag, rgt |
| $c_3$ | 0, 300, -, dwn, I, E, 100 |
| $c_2$ | 0, 150, $v_4$.hash, dwn, E, 80 |
| $w_5$ | 1, 450, -, dwn, I |
| $w_4$ | 2, 550, $v_5$.hash, dwn |
| $c_1$ | 0, 150, $v_3$.hash, dwn, E, 80 |
| $w_3$ | 2, 700, -, dwn, I |
| $w_2$ | 3, 850, $v_2$.hash, rgt |
| $w_1$ | ∞, 850, $v_1$.hash, dwn |

A proof Node (a tuple in proof vector) contains the following:

- Level
- Rank
- Hash of neighbor not included in the proof vector
- Direction of the next proof node relative to this one
- Intersection flag
- End flag
- Data length

Figure 15: An output of a multiProof algorithm.

### 5.3.2 Verifying Variable-size Multi-block Updates

When the *multiUpdate* algorithm is used at the server side of the FlexDPDP protocol, it produces a proof vector, in which all affected nodes are included, and a hash value, which corresponds to the root of the FlexList after all of the update operations are performed. Our solution to verify such an update

consists of four parts. First, **the multi proof is verified** via both FlexList verification and tag verification. Second, we **construct a temporary FlexList** that contains the parts necessary for the updates. Third, we **do the updates** as they are, at the client side. The resulting temporary FlexList has the root of the original FlexList at the server side after performing all updates correctly. Fourth and last, we **check** if the **new root** we calculated is the same as the one sent by the server. If they are the same, we accept and update the meta data kept at the client.

**Constructing a temporary FlexList out of a multi proof:** Building a temporary FlexList is giving the client the opportunity to use the regular FlexList methods to do the necessary changes to calculate the new root. **Dummy nodes** that we use below are the nodes that have some values set and are **never subject to recalculation**.

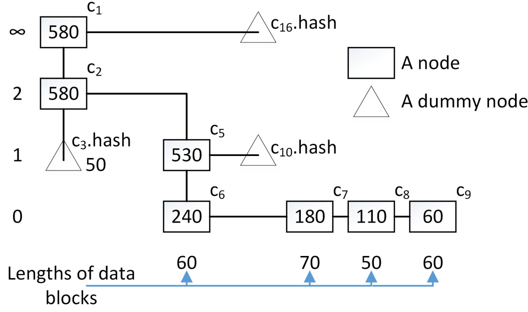


Figure 16: The temporary FlexList generated out of the proof vector in Figure 15. Note that node names are the same with Figure 3.

We explain Algorithm A.17 using the proof vector given in Figure 15. The output of the algorithm given the proof vector is the temporary FlexList in Figure 16. First, the algorithm takes the proof node for $c_1$, generates the root using its values and adds the dummy after, with the hash value (of $c_{16}$) stored in it, and the nodes are connected to each other depending on their attributes. The proof node for $c_2$ is used to add node $c_2$ to the below of $c_1$ and the $c_2$'s dummy node is connected to its below with rank value of 50, calculated as rank of $c_2$ minus rank of $c_5$. The next iteration sets $c_5$ as $c_2$'s after and $c_5$'s dummy node $c_{10}$ is added to $c_5$'s after. The next step is to add $c_6$ to the below of $c_5$. $c_6$ is both an end node and an intersection node, therefore we set its tag (from the tag vector) and its length values. Then we attach $c_7$ and calculate its length value since it is not in the proof vector generated by *genMultiProof* (but we have the necessary information: the rank of $c_7$ and the rank of $c_8$). Next, we add the node for $c_8$, and set its length value from the proof node and its tag value from the tag vector. Last, we do the same to $c_9$ as $c_8$. The algorithm outputs the root of the new temporary FlexList.

**Verification:** Recall that $U$ is the list of updates generated by the client. An update information $u \in U$, includes an index $i$, and if the update is an insertion or modification, a block and a tag value. The client calls *verifyMultiUpdate* (Algorithm A.18) with its meta data and the outputs $P$, $T$, $M$ of *multiUpdate* from the server. If *verifyMultiProof* returns accept, *buildTemporaryFlexList* is called with the proof vector $P$. The resulting temporary FlexList is ready to handle updates. The updates are performed without the hash calculations first, and then all the necessary hashes are calculated at once. (Note that 'without hash calculations' does *not* mean 'without rank calculations'. They are necessary and less time consuming.) Last, we check if the resulting hash of the root of our temporary FlexList is equal to the one sent by the server. If they are the same, we accept and update the client's meta data.

# 6 Security Analysis

In this section, we prove security of FlexDPDP, and emphasize the importance of authenticating data length values. Note that within a proof vector, all nodes that are marked with the end flag E contain the length of their associated data. These values are used to check if the server indeed sent the proof of the blocks corresponding to the challenged indices. A careless implementation may not consider the authentication of the length values. To demonstrate the consequence of *not authenticating* the length values, we use Figure 12 and Figure 13 as an example.

Suppose that the client challenges the server on the indices {170, 400, 500, 690} that correspond to nodes $c_1$, $v_4$, $c_3$, and $c_4$, respectively. Assume that the server finds out that he does not possess $v_4$ anymore, and therefore, instead of that node, he tries to deceive the client by sending a proof for $c_2$. The

proof vector will be the one illustrated in Figure 13 with a slight change done to deceive the client. The change is done to the fourth entry from the top (the one corresponding to $c_2$): Instead of the original length 80, the server puts 105 as the length of $c_2$. The verification algorithm (without authenticated length values) at the client side will accept this fake proof as follows:

- The block sum value and the tags get verified since both are prepared using genuine tags and blocks of the actual nodes. The client cannot realize that the data of $c_2$ counted in the block sum is not 105 bytes, but 80 bytes instead. This is because the largest challenged data (the data of $c_4$ of length 110 in our example) hides the length of the data of $c_2$.
- Since the proof vector contains genuine nodes (though not necessarily all the challenged ones), when the client uses *verifyMultiProof* algorithm on the proof vector from Figure 13, the hash check on line 32 of Algorithm A.13 will be passed.
- The client also checks that the proven nodes are the challenged ones by comparing the challenge indices with the reconstructed indices by "$\forall$a, $0 \leq a \leq n$ , $0 \leq C_a - V_a < endnodes_{n-a}.length$" (on line 32 of Algorithm A.13). This check will also be passed because:
  - $c_1$ is claimed to start at index 150 and contains 80 bytes, and hence includes the challenged index 170 (verified as $0 < 170 - 150 < 80$).
  - $c_2$ is claimed to start at index 300 and contains **105** bytes, and hence includes the challenged index 400 (verified as $0 < 400 - 300 < 105$).
  - $c_3$ is claimed to start at index 450 and contains 100 bytes, and hence includes the challenged index 500 (verified as $0 < 500 - 450 < 100$).
  - $c_4$ is claimed to start at index 640 and contains 110 bytes, and hence includes the challenged index 690 (verified as $0 < 690 - 640 < 110$).

There are two possible **solutions:** We may either include the authenticated rank values of the right neighbors of the end nodes in the proofs, or use the length of the associated data in the hash calculation of the leaf nodes. We choose the second solution, which is authenticating the length values, since adding the neighbor node to the proof vector also adds a tag and a hash value, for each challenged node, to the communication cost.

**Lemma 1.** *If there exists a collision resistant hash function family, FlexList is an authenticated dictionary.*

*Proof.* The only difference between FlexList and RBASL is the calculation of the rank values at the leaf levels. All rank values, which are used in the calculation of the start indices of the challenged nodes, are used in hash calculations as well. Therefore, both *length* and *rank* values contribute to the calculation of the hash value of the root. To deceive the client, the adversary should fake the rank or length value of at least one of the proof nodes. By Theorem 1 of [45], if the adversary sends a verifying proof vector for any node other than the challenged ones, we can break the collision resistance of the hash function, using a simple reduction. The reduction will keep a local copy of the data structure, and if the adversary manages to output a proof for something that is different from the local copy, then the reduction would output the adversary's output together with the non-matching part in the local copy as the collision. Since previous papers on authenticated data structures provided similar proofs, we do not extend our discussion. Therefore, we conclude that FlexList protects the integrity of the tags and data lengths associated with the leaf-level nodes. □

The tag verification protects the integrity of the data itself, based on the factoring assumption, as shown by DPDP [25]. Combining this with Lemma 1 concludes security of FlexDPDP.

**Theorem 1.** *If the factoring problem is hard and a collision resistant hash function family exists, then FlexDPDP is secure.*

*Proof.* Consider the proof of Theorem 2 in [25]. Replacing Lemma 2 in [25] used in that proof with our Lemma 1 yields an identical challenger, and shows the validity of our theorem. □

# 7 Performance Analysis

We developed an implementation of an optimized FlexList (on top of our optimized skip list and authenticated skip list implementations[3]). We then used this optimized FlexList implementation for developing FlexDPDP secure cloud storage system with multi-prove, multi-verify, and multi-update capabilities. We used C++ with the aid of the *Cashlib* library [41, 15] for cryptography and the *Boost Asio* library [13] for network programming. The local experiments were conducted on a 64-bit machine possessing 4 Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz CPU (only one core is active, except for parallel build tests), 16GB of memory and 16MB of L2 level cache, running Ubuntu 12.04 LTS. As security parameters, we used 1024-bit RSA modulus, 80-bit random numbers, and SHA-1 hash function; overall resulting in an expected security of 80-bits. All our results are the average of 10 runs. The tests include I/O access time, where **each block of the file is kept on the hard disk drive separately**, unless stated otherwise. The size of a FlexList is suitable to keep it in RAM, together with the tags but without the file blocks.

For energy efficiency tests, we used Watts up Pro meter, which measures the total energy consumption of the connected device. We conducted the tests and took their energy consumption measurements. Then, we measured the average energy cost for the idle time when no tests were taking place. The difference between these two measurements were used for the results.

In this section, our extensive analysis results of FlexList and FlexDPDP performances, deployment on the PlanetLab distributed platform, comparison with DPDP and static cloud storage on the PlanetLab are discussed.

## 7.1 Core FlexList Algorithm Performance

### 7.1.1 Unnecessary Nodes and Links Optimization

A core optimization in a FlexList is done in terms of the structure. Our optimization of removing unnecessary links and nodes results in $50\%$ fewer nodes and links on top of the leaf nodes, which are always necessary since they keep the file blocks.

Figure 17 shows the number of links and nodes used before and after optimization. The expected number of nodes in a regular skip list is $2n$ (where $n$ represents the number of blocks): $n$ leaf nodes and $n$ non-leaf nodes [49]. As described in Section 3, each non-leaf node makes any left connection below its level unnecessary. Therefore, in expectation, half of the non-leaf level $after$ links are unnecessary. Since there are $n/2$ non-leaf level unnecessary links, in expectation, this means that there are $n/2$ non-leaf level unnecessary nodes as well, according to the unnecessary node definition (Section 3). Hence, there are $n - n/2 = n/2$ non-leaf *necessary nodes*. Since each necessary node has 2 links, in total there are $2 * n/2 = n$ *necessary links* above the leaf level. That is why there is an

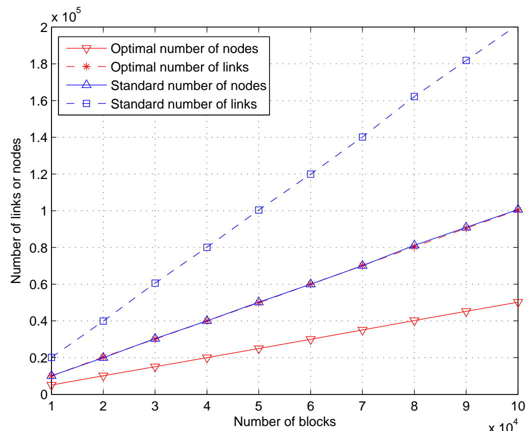


Figure 17: The number of nodes and links used on top of leaf level nodes, before and after optimization.

[3] Recall that RBASL is now a special case of FlexList, and thus needs no separate implementation.

overlap between the standard number of non-leaf nodes ($n$) and the optimal number of the non-leaf links ($n$) in Figure 17. Therefore, with this optimization **we eliminated approximately 50% of all nodes and links** above the leaf level.

#### 7.1.2 buildFlexList Performance

We presented a novel algorithm for building a FlexListefficiently. Instead of performing $n$ insertions that would require $O(n \log n)$ total time, our approach only requires $O(n)$ time in total, assuming the underlying data is already sorted, which is the case in the cloud storage scenario. Figure 18 demonstrates the energy consumption and time ratios. The time ratio is calculated by dividing the time spent for the building the FlexList via insertion (in sorted order) by the time needed by the *buildFlexList* algorithm. The same equation is applied to the energy consumption ratio calculation. Note that even though the insertions are already in sorted order, the regular insertion algorithm cannot benefit from this fact, whereas our novel *buildFlexList* algorithm is the first of its kind to make us of this fact. In these energy and time ratio experiments, we do not take into account the disk access time, therefore there is no delay for I/O switching. As expected, *buildFlexList* algorithm logarithmically outperforms the regular insertion method. Moreover, in our *buildFlexList* algorithm, the expensive hash calculations are performed only once for each node in the FlexList. As an example, the *buildFlexList* algorithm reduced the time to build a FlexList for a **file of size 400MB** with 200000 blocks **from 12 seconds to 2.3 seconds** and for a **file of size 4GB** with 2000000 blocks **from 128 seconds to 23 seconds**.

#### 7.1.3 Parallel buildFlexList Performance

Figure 19 shows the *parallel buildFlexList* execution time as a function of the number of cores employed. The case of one core corresponds to the regular *buildFlexList* function. From 2 cores to 24 cores, the time spent by our *parallel buildFlexList* function is measured. Notice the speed up where parallel build reduces the time to build a FlexList of 4 million blocks **from 240 seconds to 30 seconds on 12 cores** (including I/O time). The speedup values are reported in Figure 20 where T stands for the time taken when a single core is used, and $T_p$ stands for the time taken with $p$ number of cores used. The more sub-tasks created, the more time is required to divide the big task into parts and to combine them. We see that a FlexList of 100000 blocks does not get improved as much, since the sub tasks are getting smaller and the overhead of thread generation starts to surpass the gain of parallel operations. Starting from 12 cores, we observe this side effect for all sizes. For 500000 blocks (e.g., 1GB file) and larger FlexLists, **speed ups of 6 and 7.7 are observed on 8 and 12 cores**, respectively.

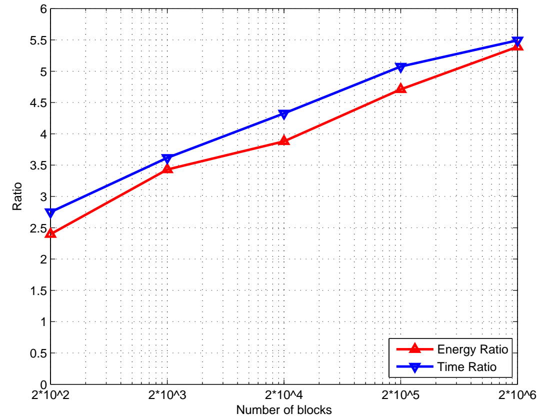


Figure 18: Time and energy efficiency ratios of our buildFlexList algorithm against insertions.

### 7.2 FlexDPDP Performance

#### 7.2.1 Effect of Block Size

Figure 21 shows the server proof generation time for FlexDPDP as a function of the block size for different file sizes of 16MB, 160MB, and 1600MB. The number of challenged blocks is fixed at 460 as suggested in PDP [4]. As shown in the figure, with the increase in block size, the time required for the proof generation increases, since with a higher block size, the block sum generation takes more time.

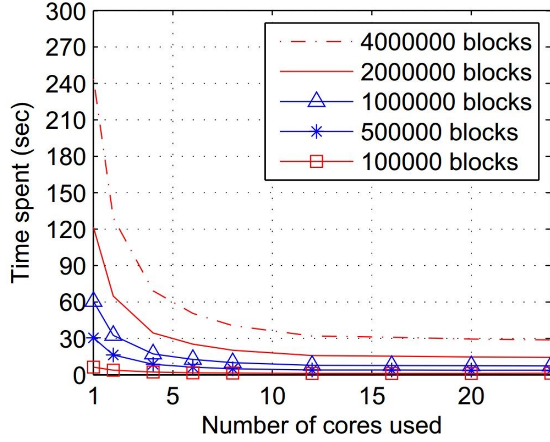


Figure 19: Time spent while building a FlexList from scratch.

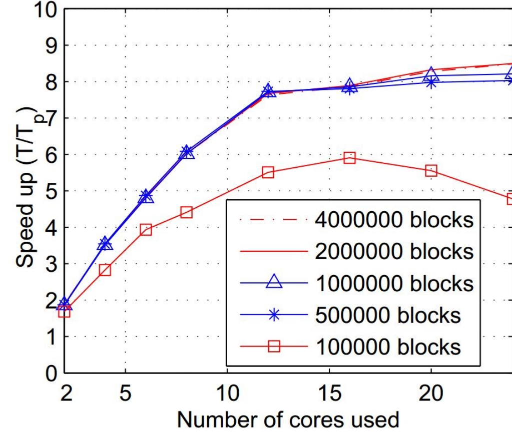


Figure 20: Speedup values of buildFlexList function with multiple cores.

However, with extremely small block sizes, the number of nodes in the FlexList become so large that it dominates the proof generation time. Since 2KB block size worked best for various file sizes, our other tests employ 2KB blocks. These 2KB **blocks are kept on the hard disk drive**; on the other hand, the **FlexList nodes** are much smaller and **are kept in RAM**. While we observed that the *buildFlexList* algorithm runs faster with bigger block sizes (since there will be fewer blocks), the creation of a FlexList happens only once. On the other hand, the proof generation algorithm runs periodically depending on the client; therefore, we chose to optimize our block size for its running time. Note that a FlexList can handle varying block sizes, yet, there is an average target block size even in a FlexList. Without such a target average size, it is possible to make a huge update efficiently once, but then later smaller updates will keep splitting that huge chunk, becoming inefficient themselves. Thus, 2KB can be interpreted as the average block size.

### 7.2.2 Multi-Proof Generation Performance

The performance of our optimized implementation of the proof generation mechanism is evaluated in terms of communication and computation. We take into consideration the case where the client wishes to detect, with more than 99% probability, if more than 1% of her 1GB data is corrupted, by challenging 460 blocks; the same scenario employed in PDP and DPDP [4, 25]. In the experiments, we used a FlexList with 500,000 nodes, where the block size is 2KB. Figure 22 shows the ratio of the unoptimized proofs over our optimized multi-proofs in terms of the **FlexList proof** size and computation, as a function of the number of challenged nodes. The unoptimized proofs correspond to proving each block separately, instead of using our *genMultiProof* algorithm for proving all of them at once. Our multi-proof optimization results in 40% **computation and** 50% **communication gains**, corresponding to FlexList proofs being up to **1.75 times as fast and 2 times as small**.

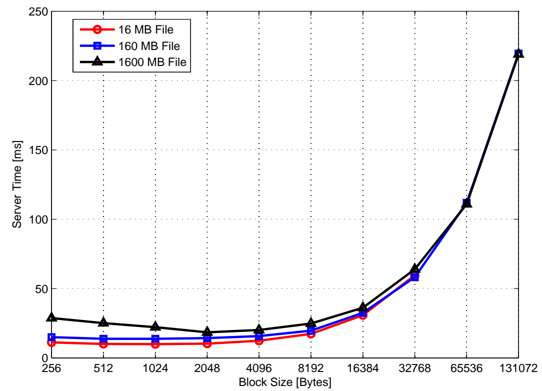


Figure 21: Server proof generation time for 460 random challenges as a function of the block size for various file sizes.

In Figure 22, the gains in a **FlexDPDP proof** size and computation time are also shown. With our optimizations, we observe gains of about **35% and 40% for the overall computation and communication**, respectively, corresponding to proofs being up to **1.60 times as fast and 1.75 times as small**. The whole proof consists of 213KB FlexList proof, 57KB of tags, and 2KB of block sum. Thus, for 460 challenges as suggested by PDP [4], we obtain a decrease in total **proof size from 485KB to 272KB**, and the computation time is **reduced from 19ms to 12.5ms** by employing our *genMultiProof* algorithm.

Note that, we could have employed a compression algorithm (e.g., zip) on top of the unoptimized proofs to eliminate duplicates in the proof, but it does not perfectly handle the duplicates, and more importantly, our algorithm also provides efficiency gains in terms of computation (whereas computing an unoptimized proof and then compressing would require even more time at both the server and client sides). Yet, if using the bandwidth efficiently is the main goal, we suggest applying compression on top of our optimal proof.

We further tested the performance of the *genMultiProof* algorithm in terms of energy efficiency. The time and energy ratios for the *genMultiProof* algorithm are shown in Figure 23. We tested the algorithm for different file sizes varying from 4MB to 4GB (where the block size is 2KB, and thus the number of blocks increases with the file size). In the *constant* scenario, we applied the same challenge size of 460 for all file sizes. Our results show a relative decline in the performance of the *genMultiProof* as the number of blocks in the FlexList increases. Because, as the number of blocks in the FlexList grows, the number of repeated proof nodes in the proof decreases (the constant number of challenged indices are distributed further away, on average, with fewer overlapping nodes along their search paths). In the *proportional* scenario, number of challenges is determined proportional to the number of blocks in the file. 5, 46, and 460 challenges are used for 20000, 200000, and 2000000 blocks, respectively. The results show a relative incline in the performance of *genMultiProof* for the proportional number of challenges. The algorithm provides an efficiency gain in the computation time in comparison to the generating each proof individually. We also observed that the energy efficiency closely follows the computational gains.

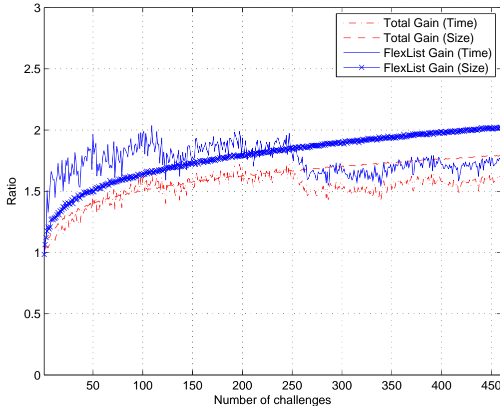


Figure 22: Performance gain graph (460 single proofs / 1 multi proof for 460 challenges).

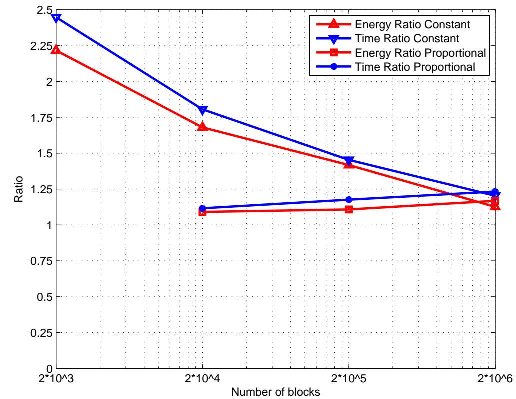


Figure 23: Time and energy ratios of the *genMultiProof* algorithm.

### 7.2.3 Server-Side Multi-Update Operations

Results for the core FlexList methods (insert, remove, modify) with and without the hash calculations, for various number of blocks, are shown in Figure 24. Even with the I/O time, **the operations with the hash calculations take 10 times more than the simple operations** on a 4GB file (i.e., 2000000 nodes). The hash calculations in an update take 90% of the time spent for an update operation. Therefore, **doing hash calculations only once for multiple updates** in the *performMultiUpdate* algorithm provides **25%**

**time and space efficiency on the verifiable update operations when the update is 20KB, and this gain increases up to 35% with 200KB updates**.

The time spent for an update at the server side for various size of updates is shown in Figure 25. Each update is an even mix of *modify*, *insert*, and *remove* operations. If the update locality is high, meaning the updates are on consecutive blocks (a diff operation generates several modifies to consecutive blocks followed by a series of remove if the new data is shorter than the old data, or a series of inserts otherwise), on a FlexList for a 1GB file, **the server time for 300 consecutive update operations (a 600KB update) is decreased from 53ms to 13ms**.

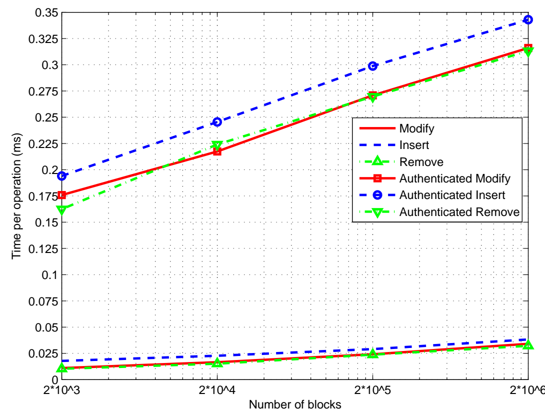


Figure 24: Time spent for an update operation in FlexList with and without hash calculations.

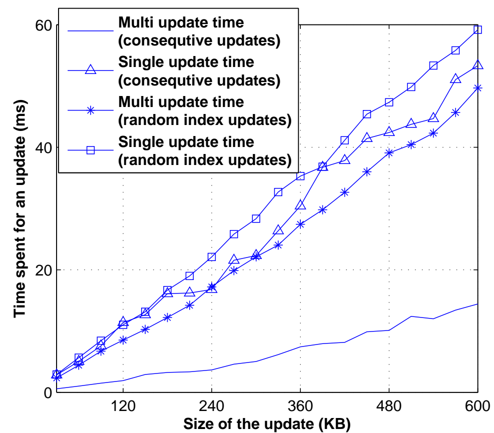


Figure 25: Time spent on performing multi updates against series of single updates.

### 7.2.4 Client-Side Multi-Update Operations

For the server to be able to use the *multiUpdate* algorithm, the client should be able to verify multiple updates at once. Otherwise, each single update verification would require a root hash value after that specific update, and thus all hash values on the search path of the update should be re-calculated each time. Also, each update proof should include a FlexList proof alongside them. Verifying multiple updates at once not only diminishes the proof size but also provides time improvements at the client side.

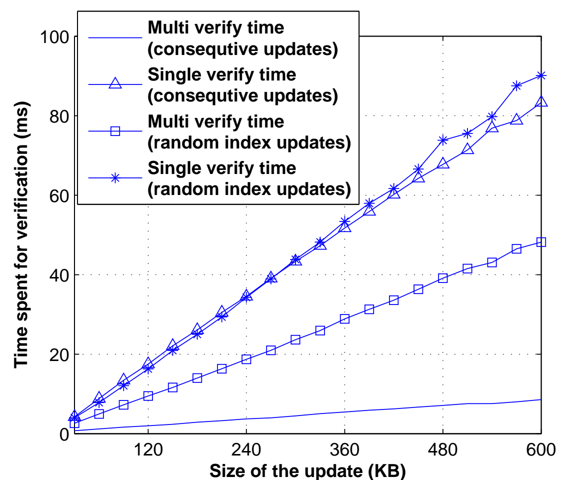


Figure 26: MultiVerify of an update against standard verify operations.

Figure 26 shows that a *verifyMultiUpdate* operation is faster at the client side when compared to verifying all the proofs one by one. We tested two scenarios: one is for the updates randomly distributed along the FlexList, and the other is for the updates with high locality. The client verification time is highly improved. For instance, with a 1GB file and a 300KB consecutive update, **verification at the client side was reduced from 45ms to less than 5ms**. With random updates, the multi-verification is still 2 times faster.

### 7.3 Real Usage Performance Analysis

We deployed the FlexDPDP implementation on the world-wide network testbed PlanetLab, and chose a PlanetLab node in Germany[4], which has two Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz (IC2), and 48Mbit upload and 80Mbit download speed, as the server. Our tests are on a 1GB file, which is divided into blocks of 2KB, having 500000 nodes (for each client). The throughput is defined as the number of queries the server can reply in a second. Our results are the average of 50 runs on the PlanetLab with randomly chosen 50 clients from all over Europe.

#### 7.3.1 Challenge queries on the PlanetLab

As shown in Figure 27, we measured two metrics: the whole time spent for a challenge-proof interaction at the client side and the throughput of the server. The maximum throughput of the server is around 21 multi-proofs per second. When the server limit is reached, we observe a slowdown on the client side where the response time increases from around 500 ms to 1250 ms. Given that preparing a proof of size 460 using the IC2 processor takes 40ms using *genMultiProof* on a single core, we conclude that the bottleneck is not the processing power. The challenge queries are solely a seed, thus the download speed is not the bottleneck either. A multi-proof for 460 blocks has an average size of 272KB; therefore, to serve 21 clients in a second, a server needs 47Mbit upload speed, which seems to be the bottleneck in this experiment. The more we increase the upload speed, the more clients the server can serve.

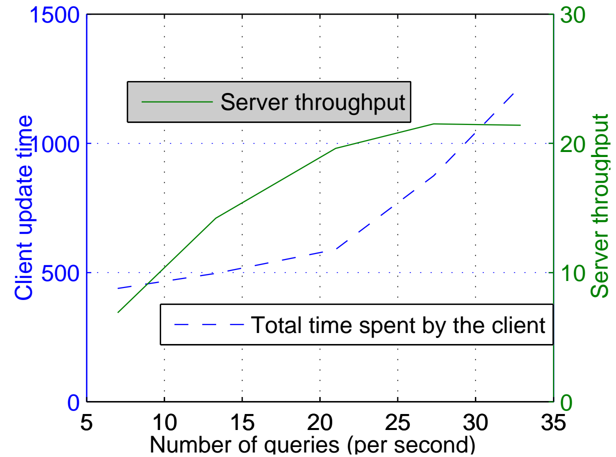


Figure 27: Clients challenging their data. Two lines present: first, server throughput in count per second and second, whole time for a challenge query of FlexDPDP, in ms.

#### 7.3.2 Update queries on the PlanetLab

We perform our analysis using the same metrics, the throughput of the server (Figure 28) and the time spent at the client side (Figure 29), for updates of size ∼20KB and ∼200KB. We test the behavior of the system by varying the query frequency, the update size, and the update type (consecutive or randomly selected blocks). Table 1 shows the measurements for each update type.

As shown in Figure 28, the server can reply ∼45 updates of size 20KB each and ∼8 updates of size 200KB each per second. Results in Figure 29 also approve that the server is loaded by an increase in time of a client getting served. From Figure 27, we conclude that the bottleneck for replying update queries is not the upload speed of the server, since a randomly distributed update of size 200KB needs a 70KB proof, and 8 proofs per second use just 4.5Mbit of the upload bandwidth, and a randomly distributed update of size 20KB needs a proof of size 11KB, and 45 proofs per second use only 4Mbit of upload bandwidth. Table 1 shows the proof generation times at the server side and the proof sizes. 30ms per 200KB random update operation is required, so a server may answer up to 110-120 queries per second with the IC2 processor. Similarly, 10ms per 20KB random update operation is required, thus a server can reply up to 300 such queries per second. Therefore, the bottleneck is not the processing power either. Eventually the amount of queries of a size a server can accept per second is limited, even though the download bandwidth does not seem to be loaded up. But, note that the download speed is checked with a single source and a continuous connection. When a server keeps accepting new connections, the

[4]planetlab1.informatik.uni-wuerzburg.de

end result is different. This was not a constraint in answering challenge queries since a challenge is barely a pseudorandom seed to show the server which blocks are challenged. In our setting, there is one thread at the server side which accepts a query and creates a thread to reply it. We conclude that the bottleneck is the server query acceptance rate of our implementation. These results indicate that with a distributed and replicated server system (e.g., [28]), a server using our FlexDPDP scheme may achieve better throughput.

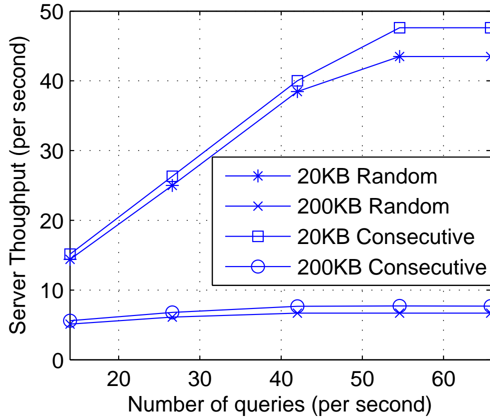


Figure 28: Server throughput versus the frequency of the client queries.

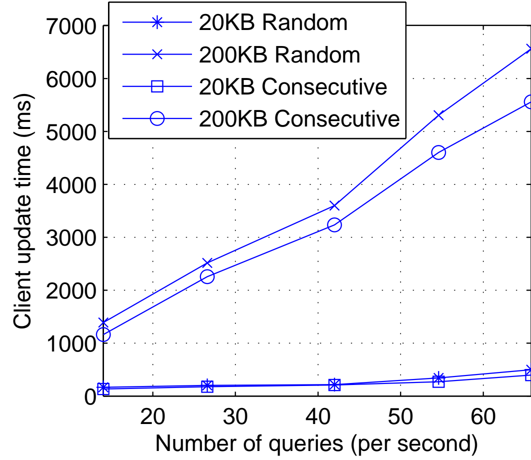


Figure 29: A client's total time spent for an update query (from sending the update till verifying the proof).

| Update size and type | Server proof generation time | Corresponding proof size |
|---|---|---|
| 200KB (100 blocks) randomly dist. | 30ms | 70KB |
| 20KB (10 blocks) randomly dist. | 10ms | 11KB |
| 200KB (100 blocks) consecutive | 7ms | 17KB |
| 20KB (10 blocks) consecutive | 6ms | 4KB |

Table 1: Proof time and size for various type of updates.

### 7.3.3 Update queries on real SVN traces

We have conducted an analysis on our SVN server, where there exist 350MB of data that we have been using for 2 years. We examined a sequence of 627 commits and provide results for an average usage of a commit function by means of the **update locality**, the **update size** being sent through the network, and the updated **number of blocks**.

We consider the directory hierarchy proposed in [25], where the idea is to set the root of each file's FlexList as the leaf nodes of a dictionary used to organize the files. More specifically, we can use our optimized authenticated skip list implementation as the dictionary whose leaves are the roots of the FlexLists, one per file in our SVN repository. We observe that the update locality of the commits is very high. More than 99% of the updates in a single commit occur in the same folder, and thus do not affect most parts of the directory. Moreover, 27% of the updates are consecutive block updates on a single field of a single file.

With each commit, an average update of size 77KB is sent, where only 2.7% of all commits have size greater than 200KB and a huge majority (85%) of all commits have size less than 20KB. These sizes refer to the amounts sent through the network. Besides, an analysis on 3 public SVN repositories was conducted by Erway et al. [25], indicating that the average update size is 28KB. This is the reason that in our experiments above on the PlanetLab we chose 20KB (to show general usage) and 200KB

(to show big commits) as the size sent for a commit call. The **average number of blocks affected per commit** provided in [25] is 13, and **is 57.7** in our SVN repository. Both **show the necessity of efficient multiple update operations**. We also observe the size variation of the commits and see that the **greatest common divisor of the size of all commits is 1 byte**, as expected. Thus we conclude that **fixed block sized rank-based authenticated skip list is not applicable to such a cloud storage scenario**.

### 7.4 Comparison with DPDP

As explained before, since DPDP scheme of Erway et al. [25] uses RBASL, it only supports working with a fixed block size. However, our analysis demonstrates that the block size common to all updates on a real SVN deployment is 1 byte, which is an impractical block size. For example, if a block size of 1 byte is used for a 1GB file, the corresponding RBASL will have more than 1 billion blocks. But, remember that FlexList has a flexible block size. Therefore, if the initial block size is set as, for example, 2KB, then the corresponding FlexList will have only 500 thousand blocks. Thus, it is clear that the RBASL-based DPDP will be much slower than FlexDPDP.

On the other hand, using the same 2KB block size with RBASL would mean that either the updates will need to be padded, or to keep the block size fixed, an updated block will affect all the other blocks to the right of it. If we consider the second option, it is obvious that **an RBASL will be, on average, $\mathbf{n/2}$ times slower than a FlexList with the same number of blocks**. For $n = 500000$, this is an intolerable performance loss.[5] On the other hand, if we consider the first option, then such a DPDP scheme can only be used if the client also keeps a local copy of the RBASL, and keeps it up-to-date with each update, wasting both space and time. Such a copy is necessary, since otherwise the client has no way of knowing, before contacting the server, which block contains the byte that it wants. Since most diff algorithms are byte-based, if no local RBASL is kept, each update would require first obtaining the byte-to-block conversion from the server together with its proof, and then performing an update with the padding. There are known problems with padding, including the extra space requirement. Furthermore, the use of padding in this cloud storage scenario, at the very least, **duplicates** the computation and communication done for updates, as explained above. Note that, with flexible block size, FlexList never faces such problems, since a FlexList proof already contains authenticated length values of the blocks, and the client can verify that the byte indices output by the diff algorithm correspond to the blocks updated by the server.

Therefore, we proposed FlexList as an elegant solution to all these problems. Rather than adding more space or time requirements, we show that indeed **FlexList reduces both time and space requirements, compared to RBASL and similar data structures**. Furthermore, all our optimizations presented, including the first-known $O(n)$ build that can be parallelized, and multi-prove and multi-update capabilities, can be applied on top of RBASL and authenticated skip lists as well, since RBASL is indeed a special case of FlexList. Thus, an efficient way to implement such a secure cloud storage service is through FlexList and FlexDPDP. Next, we show that **FlexDPDP, though being dynamic, has almost the same performance as the best-known static scheme**.

### 7.5 Comparison with Static Cloud Storage on the PlanetLab

We compare static PDP [4] with our dynamic FlexDPDP. PDP server computes the sum of the challenged blocks, and performs the multiplication and exponentiation –by random challenge exponents– of their tags. FlexDPDP server only computes the sum of the challenged blocks and a FlexList proof, but not the multiplication and exponentiation of their tags, which are done at the client side. In such a scenario, the FlexDPDP server outperforms such a naive PDP server, since the multiplication and exponentiation of tags by the PDP server are expensive cryptographic operations that take much longer than the FlexList proof generation in FlexDPDP. This result is surprisingly in contrast to the fact that PDP proofs take

[5] We actually prepared a graph for this comparison, but it indeed almost exactly shows $n/2$ performance improvement ratio in all our tests with random updates, and hence we chose not to waste space showing such an obvious experiment result.

$O(1)$ time whereas FlexDPDP proofs require $O(\log n)$ time, mainly due to this big difference in the constants in the Big-Oh notation.

We note that it is possible for PDP to be implemented by the server sending the tags to the client and the client computing the multiplication and exponentiation of the tags. We call this modified version of the PDP protocol PDP*. Even though the proof size grows in PDP*, the PDP* server can respond to challenges faster than the FlexDPDP server, presenting a computation-communication trade-off for the server. Therefore, we realize that **where to handle the multiplication and exponentiation of tags is an important implementation decision** for PDP.

We deployed FlexDPDP, together with the original and modified PDP versions, on the PlanetLab. A node on the PlanetLab has minimum requirements of having 6x Intel Xeon E5 cores @ 2.2GHz processor, 24 GB of RAM, and 2TB shared hard disk space. The nodes are also required to have minimum of 400kbps of bi-directional bandwidth to the Internet [46]. As a central point in Europe, we chose a node in Berlin, Germany[6] as the server. We measured the whole time spent for one challenge, starting from the time the client starts to generate the challenge, until the time that the client receives and verifies the proof (see Table 2). We moved our client location and tested serving a *close range client* in Munich, Germany[7], a *mid-range client* in Koszalin, Poland[8], and a *distant client* in Lisbon, Portugal[9]. We used a single core at each side. The protocols are run on a 1GB file, which is divided into blocks of 2KB, having 500000 nodes.

As shown on Table 2, we conclude that using 6 cores (usual core count in PlanetLab nodes), a **PDP server can answer 14.5 queries per second** whereas a server using **PDP* or FlexDPDP can serve 462 queries or 155.5 queries per second**, respectively. We discern that, for the server, tag exponentiation and multiplication are the most time consuming tasks in preparing a proof. It is clear that to increase the server throughput, tag exponentiation and multiplication should be delegated to the client. This delegation increases the total time spent by the client, since the tags should be sent over the network now. However, the outcome is the dramatic increase in the server throughput. Note that, when one considers the total time a client spends for sending a challenge, obtaining the proof, and verifying it, **the overhead of being dynamic (FlexDPDP vs. PDP*) is around 40 to 90 ms, which is a barely-visible difference for a real-life application (especially considering that the whole process takes on the order of a second)**.

| | PDP | PDP* | FlexDPDP |
|---|---|---|---|
| Local Server Computation | 413.19 | 12.97 | 38.60 |
| Close Client Total | 466.82 | 557.49 | 649.11 |
| Mid-range Client Total | 496.856 | 714.47 | 874.63 |
| Distant Client Total | 551.376 | 986.98 | 1023.25 |

Table 2: Time spent for a challenge of size 460, in milliseconds. PDP* is the modified PDP scheme, where we send all the challenged tag values to the client individually, instead of processing them at the server.

## 8 Related Work

**Skip Lists and Other Data Structures:** Table 3 provides an overview of data structures proposed for secure cloud storage. Among the structures that enable dynamic operations, the advantage of skip list is that it keeps itself balanced probabilistically, without the need for complex balancing operations

[6] planetlab01.tkn.tu-berlin.de
[7] planetlab1.lkn.ei.tum.de
[8] ple2.tu.koszalin.pl
[9] planetlab1.di.fct.unl.pt

[49]. It offers search, modify, insert, and remove operations with *logarithmic* complexity with high probability [48]. A skip list is similar to a binary tree in the sense that the $below$ and $after$ links of a node can be thought to correspond to the *left* and *right* children. Skip lists have been extensively studied [3, 7, 22, 25, 33, 40, 47]. They are used as authenticated data structures in two-party protocols [45], in outsourced network storage [33], with authenticated relational tables for database management systems [7, 27], in timestamping systems [10, 11], in outsourced data storages [25, 31], and for authenticating queries for distributed data of web services [47].

In a skip list, not every edge or node is used during a search or update operation; for efficiency, those unnecessary edges and nodes should be omitted. Similar optimizations for authenticated skip lists were tested in [30]. Furthermore, as observed in DPDP [25] for an RBASL, some corner nodes can be eliminated to decrease the overall number of nodes. Our FlexList contains all these optimizations and further improvements, analyzed both formally and experimentally.

A binary tree-like data structure called rope is similar to our FlexList [12]. It was originally developed as an alternative to the strings; bytes can be used instead of the strings as in our scheme. Since a rope is a tree-like structure, its disadvantage is that it requires *balancing* operations. Moreover, a rope needs further structure optimizations to eliminate unnecessary nodes.

| | Hash Map (the whole file) [50] | Hash Map (block by block) [44] | PDP [4] | Merkle Tree [54] | Balanced Tree (2-3 Tree) [43, 56] | RBASL [25] | FlexList |
|---|---|---|---|---|---|---|---|
| Client Storage | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| Proof time/size | $O(n)$ | $O(1)$ | $O(1)$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| Dynamic op. capability | - (none) | - (append-only) | - (append-only) | - (worst-case $O(n)$) | + (balancing issues) | + (fixed block size) | + (fully dynamic) |

Table 3: Complexity and capability table of various data structures for provable cloud storage. $n$ is the number of blocks.

**Cloud Storage Related Work:** PDP [4] was one of the first proposals for efficient provable cloud storage. PDP does not employ a data structure for the authentication of the blocks, and is applicable to only static (append-only) storage. A later variant called Scalable PDP [6] allows a limited number of updates. Wang et al. [54] proposed the usage of Merkle tree [42], which works perfectly for the static scenario, but has balancing problems in a dynamic setting. If the insertion and deletions are mostly over a small subset of the indices, the performance can degrade down to an $O(n)$ worst-case due to an imbalanced tree. For the dynamic case, we would need an authenticated *balanced* tree such as the data structure proposed in [56], named range-based authenticated 2-3 tree [43]. Nevertheless, skip lists are much simpler data structures to implement, and more importantly algorithms for efficiently updating and maintaining authentication information have been studied in detail for the authenticated skip list [45]. Table 3 summarizes this comparison.

For improving data integrity on the cloud, some protocols [20, 35, 17, 38, 34, 39] provide Byzantium fault-tolerant storage services based on some server labor. There also exist protocols using quorum techniques, which do not consider the server-client scenarios but works on local systems such as hard disk drives or local storage [36, 29, 1, 21]. A recent protocol using quorum techniques [9] replicates the data on several storage providers to improve integrity of the data stored on the cloud; yet it also considers static data.

For dynamic provable data possession (DPDP) in a cloud storage setting, Erway et al. [25] were the

first to introduce the new data structure called the *rank-based authenticated skip list (RBASL)*, which is a special type of an authenticated skip list [33]. In the DPDP model, there is a client who wants to outsource her file and a server that takes the responsibility for the storage of the file. The client pre-processes the file and maintains meta data to verify the proofs from the server. Then she sends the file to the server. When the client needs to check whether her data is intact or not, she challenges some random blocks. Upon receipt of the request, the server generates a corresponding proof and sends it back. The client then verifies the integrity of the file using this proof. Many other static and dynamic schemes have been proposed [37, 51, 24, 18, 19, 52] including multi-server optimizations on them [14, 23, 28].

An RBASL, unlike an authenticated skip list, allows a search with the indices of the blocks. This gives the opportunity to efficiently check the data integrity using block indices as parameters in the proof and update queries in DPDP. Each node in the RBASL has a *rank*, indicating the number of the leaf-level nodes (equivalently, the number of file blocks) that are reachable from that particular node. Leaf-level nodes having null $after$ links have a rank of 1, meaning they can be used to reach themselves only. Ranks in an RBASL solve the problem of updating the block numbers in PDP [4], and are thus used to construct a dynamic system.

Nevertheless, in a realistic scenario, the client may wish to change a *part* of a block, not the *whole* block. To partially modify a particular block in an RBASL, we not only modify a specified block but also may have to change all following blocks. This means the number of modifications is $O(n)$ in the worst-case scenario for DPDP as well (see Sections 4 and 7.4).

Another dynamic provable data possession scheme employs a new data structure called a balanced update tree, whose size grows with the number of the updates performed on the data blocks [55]. Due to this property, extra rebalancing operations are required. The scheme uses message authentication codes (MAC) to protect the data integrity. Unfortunately, since the MAC values contain indices of the data blocks, they need to be recalculated with insertions or deletions. The data integrity checking can also be costly, since the server needs to send all the challenged blocks with their MAC values, because the MAC scheme is not homomorphic (see [5]).

Our proposed data structure FlexList, based on an RBASL, performs dynamic operations (modify, insert, remove) for cloud data storage, efficiently handling variable-sized updates in a provably secure manner. Furthermore, we present the first optimized construction of such a scheme, and demonstrate both theoretical and experimental improvements.

# 9 Conclusions and Future Work

The security and privacy concerns are significant obstacles towards the adoption of cloud storage [57]. With the emergence of cloud storage services, data integrity has become one of the most important challenges. Early works have shown that the static solutions with optimal complexity [4, 51], and the dynamic solutions with logarithmic complexity [25] are within reach. However, a DPDP [25] solution is not applicable to real life scenarios since it supports only fixed block size and therefore lacks flexibility on the data updates, while the real life updates are not multiples of a meaningful constant block size. We have extended earlier studies in several aspects, provided a new data structure (FlexList) and its optimized implementation for use in the cloud data storage. A FlexList efficiently supports variable block sized dynamic provable updates, and we showed how to handle multiple proofs and updates at once significantly improving scalability. We also studied energy efficiency of FlexList and FlexList-based cloud storage, FlexDPDP. We showed, for the first time in the literature, how to build such a data structure from scratch in $O(n)$ time, instead of $O(n \log n)$ time. We also proposed how to parallelize such an authenticated structure. As future work, we plan to further study parallelism on multi- proofs and updates, and also aim to extend our system to peer-to-peer storage settings.

## Acknowledgements

We would like to thank Ozan Okumuşoğlu at Koç University for his contribution on testing and debugging, working on implementation of server-client side of the project and verification algorithms. We also acknowledge the support of TÜBİTAK (the Scientific and Technological Research Council of Turkey) under project numbers 112E115, 109M761, European Union COST Actions IC0804 and IC1306, Türk Telekom, Inc. under grant 11315-06, and Koç Sistem, Inc. A preliminary version of this work was presented in ICC conference [26].

## References

[1] I. Abraham, G. Chockler, I. Keidar, and Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 2006.

[2] M. AdelsonVelskii and E. Landis. An algorithm for the organization of information. *Defense Technical Information Center*, 146:263–266, 1963.

[3] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *ISC*, 2001.

[4] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM CCS*, 2007.

[5] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *ASIACRYPT*, 2009.

[6] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm*, 2008.

[7] G. D. Battista and B. Palazzi. Authenticated relational tables and authenticated skip lists. In *DBSec*, 2007.

[8] R. Bayer. Symmetric binary b-trees: Data structure and maintenance. *Acta informatica*, 1:290–306, 1972.

[9] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage*, 9(4), Nov 2013.

[10] K. Blibech and A. Gabillon. Chronos: an authenticated dictionary based on skip lists for timestamping systems. In *SWS*, 2005.

[11] K. Blibech and A. Gabillon. A new timestamping scheme based on skip lists. In *ICCSA (3)*, 2006.

[12] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: an alternative to strings. *Software: Practice and Experience*, 25, 1995.

[13] Boost asio library. http://www.boost.org/doc/libs.

[14] K. D. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *ACM CCS*, 2009.

[15] Brownie cashlib cryptographic library. http://github.com/brownie/cashlib.

[16] C. Cachin, I. Keidar, and A. Shraer. Trusting the cloud. *SIGACT News*, 2009.

[17] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *DSN*, DSN '06, Washington, DC, USA, 2006. IEEE Computer Society.

[18] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In *EUROCRYPT*, 2013.

[19] N. Chandran, B. Kanukurthi, and R. Ostrovsky. Locally updatable and locally decodable codes. Cryptology ePrint Archive, Report 2013/520.

[20] G. Chockler, R. Guerraoui, I. Keidar, and M. Vukolic. Reliable distributed storage. *Computer*, 2009.

[21] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *ACM PODC*. ACM Press, 2002.

[22] S. A. Crosby and D. S. Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM TISSEC*, 2011.

[23] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In *ICDCS*, 2008.

[24] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC*, 2009.

[25] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM CCS*, 2009.

[26] E. Esiner, A. Küpçü, and O. Özkasap. Analysis and optimization on flexdpdp: A practical solution for dynamic provable data possession. In *Intelligent Cloud Computing (ICC) conference*, 2014.

[27] M. Etemad and A. Küpçü. Database outsourcing with hierarchical authenticated data structures. In *ICISC*, 2013.

[28] M. Etemad and A. Küpçü. Transparent, distributed, and replicated dynamic provable data possession. In *ACNS*, 2013.

[29] E. Gafni and L. Lamport. Disk paxos. *Distrib. Comput.*, 16(1):1–20, Feb. 2003.

[30] M. T. Goodrich, C. Papamanthou, and R. Tamassia. On the cost of persistence and authentication in skip lists. In *Proceedings of the 6th international conference on Experimental algorithms*, 2007.

[31] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *ISC*, 2008.

[32] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Johns Hopkins Information Security Institute, 2001.

[33] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA*, 2001.

[34] G. Goodson, J. Wylie, G. Ganger, and M. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *DSN*, DSN '04, 2004.

[35] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead byzantine fault-tolerant storage. In *ACM SOSP*, SOSP '07. ACM, 2007.

[36] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 1998.

[37] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *ACM CCS*, 2007.

[38] B. Liskov and R. Rodrigues. Tolerating byzantine faulty clients in a quorum system. *IEEE 32nd International Conference on Distributed Computing Systems*, 2006.

[39] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distrib. Comput.*, 1998.

[40] P. Maniatis and M. Baker. Authenticated append-only skip lists. *Acta Mathematica*, 2003.

[41] S. Meiklejohn, C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. Zkpdl: Enabling efficient implementation of zero-knowledge proofs and electronic cash. In *USENIX Security*, 2010.

[42] R. Merkle. A digital signature based on a conventional encryption function. *LNCS*, 1987.

[43] M. Naor and K. Nissim. Certificate revocation and certificate update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, April 2000.

[44] A. Oprea, M. K. Reiter, and K. Yang. Space-efficient block storage integrity. In *NDSS*, 2005.

[45] C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *ICICS*, 2007.

[46] Planetlab node requirements. https://planet-lab.org/node/225 [Online; accessed 20-March-2013].

[47] D. J. Polivy and R. Tamassia. Authenticating distributed data using web services and xml signatures. In *In Proc. ACM Workshop on XML Security*, 2002.

[48] W. Pugh. A skip list cookbook. Technical report, University of Maryland, 1990.

[49] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 1990.

[50] T. S. J. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *IEEE ICDCS*, 2006.

[51] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, 2008.

[52] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In *ACM CCS*, 2013.

[53] P. T. Stanton, B. McKeown, R. C. Burns, and G. Ateniese. Fastad: an authenticated directory for billions of objects. *SIGOPS Oper. Syst. Rev.*, 2010.

[54] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *ESORICS*, 2009.

[55] Y. Zhang and M. Blanton. Efficient dynamic provable data possession of remote data via balanced update trees. *ASIA CCS*, 2013.

[56] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *CODASPY*, 2011.

[57] M. Zhou, R. Zhang, W. Xie, W. Qian, and A. Zhou. Security and privacy in cloud computing: A survey. In *IEEE SKG*, 2010.

# A Detailed Algorithms

## A.1 FlexList Preliminaries

Table 4 shows the notation used in the FlexList algorithms.

| Symbol | Description |
|---|---|
| $cn$ | current node |
| $pn$ | previous node, indicates the last node that current node moved from |
| $mn$ | missing node, created when there is no node at the point where a node has to be linked |
| $nn$ | new node |
| $dn$ | node to be deleted |
| $after$ | the after neighbor of a node |
| $below$ | the below neighbor of a node |
| $r$ | rank value of a node |
| $i$ | index of a byte |
| $npi$ | a boolean which is always true except in the inner loop of *insert* algorithm |
| $\sqcup_n$ | stack (initially empty), filled with all visited nodes during $search$, $modify$, $insert$ or $remove$ algorithms |

Table 4: Symbol descriptions of skip list algorithms.

**Algorithm A.1**: canGoBelow Algorithm

**Input**: current node $cn$, search index $i$
**Output**: True/False

```
1  return i < cn.below.r + 1; // if called in the insert algorithm, "+1" is not used
```

**Algorithm A.2**: canGoAfter Algorithm

**Input**: current node $cn$, search index $i$
**Output**: True/False

```
1  if i > cn.below.r then
       // if called in the isIntersection method, use ≥ above, and do not perform the
       following if statement
2      if cn.after.data.length > i - cn.below.r then
3          return false
4      return true
5  return false
```

**Algorithm A.3**: nextPos Algorithm

**Input**: $cn$, $i$, $level$, $npi$, $\sqcup_n$
**Output**: $cn$, $i$, $\sqcup_n$

```
1  while canGoBelow(cn, i) OR canGoAfter(cn, i) do
2      if canGoBelow(cn, i) AND cn.below.level ≥ level AND npi then
3          cn = cn.below
4      else if canGoAfter(cn, i) AND cn.after.level ≥ level then
5          i = i - cn.below.r// update search index
6          cn = cn.after
7      add cn to ⊔n
```

**Algorithm A.4**: createMissingNode Algorithm

**Input**: $pn$, $cn$, $i$, $level$, $\sqcup_n$
**Output**: $pn$, $cn$, $i$, $\sqcup_n$

```
1  mn = new node is created using level // rank value for mn is set to ∞ for now
   // put to correct location
2  if canGoBelow(cn,i) then
3      mn.below = cn.below
4      cn.below = mn
5  else
6      mn.below = cn.after
7      cn.after = mn
8      i = i - cn.below.r // update search index
9  pn = cn
10 cn = mn
11 add cn to ⊔n
```

**Algorithm A.5**: deleteUNode Algorithm

**Input**: $pn$, $cn$, $\sqcup_n$
**Output**: $pn$, $cn$, $\sqcup_n$

```
1  if cn.level == 0 then
2      cn.after = NIL
3  else
4      if pn.below == cn then
5          pn.below = cn.below
6      else
7          pn.after = cn.below
8      ⊔n.pop()
9      cn = pn
```

## A.2 Algorithms of FlexList Methods

**Algorithm A.6**: search Algorithm

**Input**: search index $i$
**Output**: node $cn$ at the index, stack $\sqcup_n$ of nodes on the search path

```
1  ⊔n = new empty Stack
2  cn = root
3  call nextPos // cn moves until cn.after is a tower node of the searched index. nextPos also
   adds nodes on the path to the stack ⊔n.
4  cn = cn.after
5  add cn to ⊔n
   // Loop moves cn below until the node at the leaf level
6  while cn.level ≠ 0 do
7      cn = cn.below
8      add cn to ⊔n
```

**Algorithm A.7**: modify Algorithm

**Input**: index $i$, new data $data$
**Output**: node $cn$ at the index, stack $\sqcup_n$ of nodes on the search path

```
1  ( cn, ⊔n) = search(i)
2  cn.data = data
   // For an authenticated structure, call calculateHash on the nodes in the ⊔n to
   re-compute their hashes.
```

**Algorithm A.8**: insert Algorithm

**Input**: index of insertion $i$, data
**Output**: the new leaf-level node $nn$, stack $\sqcup_n$ of search path

```
1   ⊔n = new empty Stack
2   pn = root
3   cn = root
4   level = tossCoins()
5   call nextPos // cn moves until it finds a missing node or cn.after is where nn is to be
    inserted
    // Check if there is a node where new node will be linked.  if not, create one.
6   if !CanGoBelow(cn, i) or cn.level ≠ level then
7       call createMissingNode;
    // Create new node and insert after the current node.
8   nn = new node created using level level, index i
9   nn.after = cn.after
10  cn.after = nn
11  add nn to ⊔n
    // Create insertion tower until the leaf level is reached.
12  while cn.below ≠ null do
13      if nn.after ≠ null then
14          tn = new node created using index i
15          nn.below = tn
16          nn = nn.below
17          add nn to ⊔n
18      call nextPos // Current node moves until we reach an after link that passes through the
        tower.  That is the insertion point for the new node.
        // Create next node of the insertion tower.
19      nn.after = cn.after
20      cn.after = null
21      nn.level = cn.level
        // cn becomes unnecessary as it looses its after link, therefore it is deleted
22      deteletUNode(pn, cn);
    // Done inserting, put data and return this last node.
23  nn.data = data
    // For an authenticated structure, call calculateHash on the nodes in the ⊔n to
    re-compute their hashes.
    // For FlexList or RBASL, call calculateRank on the nodes in the ⊔n to re-compute their
    ranks.
```

**Algorithm A.9**: remove Algorithm

**Input**: removal index $i$
**Output**: deleted node $dn$, stack $\sqcup_n$ of search path of the left leaf-level neighbor of the node removed

```
1   ⊔n = new empty Stack
2   pn = root
3   cn = root
4   call nextPos // Current node moves until after of the current node is the node at the top
    of deletion tower
5   dn = cn.after
    // Check if current node is necessary,if so it can steal after of the node to delete,
    otherwise delete current node
6   if cn.level = dn.level then
7       call deleteNode(cn, dn)
8       dn = dn.below; // unless at leaf level
9   else
10      call deleteUNode(pn, cn)
    // Delete whole deletion tower until the leaf level is reached
11  while cn.below ≠ null do
12      call nextPos // Current node moves until it finds a missing node
        // Create the missing node unless at leaf level and steal the after link of the node
        to delete
13      call createMissingNode
14      call deleteNode(cn, dn)
15      dn = dn.below // move dn to the next node in the deletion tower unless at leaf level
    // For an authenticated structure, call calculateHash on the nodes in the ⊔n to
    re-compute their hashes.
    // For FlexList or RBASL, call calculateRank on the nodes in the ⊔n to re-compute their
    ranks.
```

## A.3 Novel Build from Scratch Algorithm

**Algorithm A.10**: buildFlexList Algorithm

```
    Input: block list B, level list L, tag list T
    Output: root
    // H will keep pointers to leftmost tower heads at each level
1   H = new vector is created of size L_0 + 1
    // iterate for each block
2   for i = B.size − 1 to 0 do
3       pn = null
4       for j = 0 to L_i+1 do
            // Always do this for leaf level, and otherwise do only if H_j contains an
            element
5           if H_j ≠ null or j = 0 then
6               nn = new node is created with level j //if at the leaf level, link to block B_i, and tag
                T_i from nn
7               nn.below = pn
8               nn.after = H_j // Connect tower head at H_j as an after link
9               call calculateRank and calculateHash on nn
10              pn = nn
11              H_j = null // Tower head will be at a higher level
12      H_{L_i} = pn // Add a tower head to H at level L_i
13  root = H_{L_0}
14  root.level = ∞
15  call calculateHash on root
16  return root
```

The algorithm starts with the creation of the vector $H$ to hold pointers to the tower heads (line 1), where $H_i$ represents the leftmost tower head at level $i$. For the first iteration of the inner loop (lines 6-11), the node $v_1$ in Figure 8 is created, which is a leaf node with no node below. The hash and the rank values of $v_1$ are calculated. Currently, $H$ is empty; therefore there is no node at $H_0$ to connect to $v_1$ at level 0, and no new nodes are created at levels 1, 2, 3, 4. At line 12, $v_1$ is put into $H$ as $H_4$, because $v_1$ is currently the head node of the leftmost tower in the FlexList at level 4 (remember, the associated level for insertion is 4). The algorithm continues with the next block and the creation of $v_2$. $H_0$ is still empty, therefore no $after$ link for $v_2$ is set. The hash and the rank values of $v_2$ are calculated. The next two iterations of the inner loop skips the lines 6-11, because $H_1$ and $H_2$ are $null$. At line 12, $v_2$ is inserted to $H_0$. At this point, $H$ only contains two values that are not $null$: $H_0 = v_2$ and $H_4 = v_1$.

Then, $v_3$ is created and its hash and rank values are calculated. It takes the current $H_0$ as its $after$ link (i.e., line 8 sets $v_3.after = H_0 = v_2$). $H_0$ becomes $null$. The level of $v_3$ is 1, and $H_1$ is $null$, therefore we set $H_1$ to $v_3$. Next, node $v_4$ is created and there is no element at $H_0$ to connect to $v_4$. The hash and the rank values of $v_4$ are calculated, and then we set $H_0$ as $v_4$, as its level is 0. Next $v_5$ is created, its after is set (i.e., line 8 sets $v_5.after = H_0 = v_4$), and the new $H_0$ becomes $v_5$, since its level is 0 as well. When $v_6$ is created, it becomes $H_0$, taking $v_5$ as its $after$. Since $H_1$ is not $null$, we create $v_7$ at level 1 and set $v_7.after = H_1 = v_3$. The algorithm continues in this manner.

## A.4 FlexDPDP Preliminaries

Table 5 shows the notation used in the FlexDPDP algorithms.

| Symbol | Description |
|---|---|
| hash | hash value of a node |
| $rs$ | rank state, indicates the byte count to the left of current node and used to recover the index value $i$ when a roll-back to a state is done |
| $state$ | state, created in order to store from which node the algorithm will continue, contains a node, rank state, and last index |
| $C$ | challenged indices vector, in ascending order |
| $V$ | verify challenge vector, reconstructed during verification to check whether or not the proof indeed belongs to challenged blocks, in terms of indices |
| $p$ | proof node |
| $P$ | proof vector, stores proof nodes for all challenged blocks |
| $T$ | tag vector of the challenged blocks |
| $M$ | block sum, which is a combined block (see [4, 25]) |
| $\sqcup_s$ | intersection stack, stores $state$ at intersections in $searchMulti$ algorithm |
| $\sqcup_h$ | intersection hash stack, stores hash values to be used at intersections |
| $\sqcup_i$ | index stack, stores pairs of integer values, employed in $updateRankSum$ |
| $\sqcup_l$ | changed nodes' stack, stores nodes for later hash calculation, employed in $hashMulti$ |
| $start$ | start index in $\sqcup_i$ from which $updateRankSum$ should start |
| $end$ | end index in $\sqcup_i$ |
| $first$ | current index in $C$ |
| $last$ | end index in $\sqcup_s$ |

Table 5: Symbols used in our algorithms.

**Algorithm A.11**: searchMulti Algorithm

**Input**: current node $cn$ to continue from, challenge vector $C$, first index $first$ to $C$, second index $last$ to $C$, current rank state $rs$, proof vector $P$ to add to, intersection stack $\sqcup_s$ to add to
**Output**: updated current node $cn$, updated proof vector $P$, updated intersection stack $\sqcup_s$

1 $i = C_{first} - rs$ // Challenged index is calculated according to the current sub skip list root
// Create and put proof nodes on the search path to the proof vector
2 **while** *Until challenged node is included in* $P$ **do**
3 $p$ = new proof node with level $cn.level$ and rank $cn.r$
// If the node at the challenged index is found, end this branch of the proof path
4 **if** $cn.level$ = 0 *and* $i < cn.length$ **then**
5 $p.setEndFlag()$
6 $p.length = cn.length$
// Get rid of indices that are on the same challenged node
7 tempStart = $first$
8 **while** $C_{tempStart}$ + $cn.data.length$ - $i > C_{tempStart+1}$ **do**
9 $first = first$ + 1
// When an intersection is found with another branch of the proof path, save the state
10 **if** $isIntersection(cn, C, i, last_k, rs)$ **then**
//note that $last_k$ becomes $last_{k+1}$ in $isIntersection$ method
11 $p.setInterFlag()$
12 add $state(cn.after, last_k, rs$+$cn.below.r)$ to $\sqcup_s$ // Add a state for $cn.after$ to continue from there later
// Missing fields of the proof node are filled according to the link current node follows
13 **if** $(CanGoBelow(cn, i))$ **then**
14 $p$.hash = $cn.after$.hash
15 $p.rgtOrDwn = dwn$
16 $cn = cn.below$ //unless at the leaf level
17 **else**
18 $p$.hash = $cn.below$.hash
19 $p.rgtOrDwn = rgt$
20 $cn = cn.after$
// Update index and rank state values according to how many bytes at leaf nodes are passed while following the $after$ link
21 $i$ -= $cn.below.r$
22 $rs$ += $cn.below.r$
23 add $p$ to $P$

## A.5 Handling Multiple Challenges at Once

**Algorithm A.12**: genMultiProof Algorithm

**Input**: challenge vector $C$, random number vector $A$
**Output**: tag vector $T$, block sum $M$, proof vector $P$

Let $A = (a_0, \ldots, a_k)$, and state be a triplet $state$=($node, lastIndex$, $rs$)
1 $cn = root$
2 $rs$ = 0
3 $M$ = 0
4 Initialize empty $\sqcup_s$, $P$, $T$
5 add state(root, k, $rs$) to $\sqcup_s$
// Call $searchMulti$ method for each challenged block to fill the proof vector $P$
6 **for** $i = 0$ *to* $k$ **do**
7 $state = \sqcup_s$.pop()
8 $cn = searchMulti(state$.node,$C, i, state$.lastIndex,$state.rs$,$P$,$\sqcup_s)$ // $searchMulti$ adds proof nodes to $P$, adds intersection states to $\sqcup_s$, and returns the node containing index $C_i$
// Store tag of the challenged block and compute the block sum

In Figure 12, the challenge vector $C$ generated from the random seed sent by the client contains [170, 320, 470, 660] as the challenged indices. By taking this challenge vector $C$ as input, the *genMultiProof* algorithm generates the proof $P$, collects the tags into the tag vector $T$, calculates the block sum $M$, and returns all three. The algorithm starts traversing from the root ($w_1$ in the figure) by retrieving it from the intersection stack $\sqcup_s$ at line 7. Then, in the loop, we call *searchMulti*, which adds the proof nodes for $w_1$, $w_2$, $w_3$ and $c_1$ to the proof vector $P$ (see bottom four elements in Figure 13). The state of node $w_4$ is saved in the stack $\sqcup_s$ as it is the $after$ of an intersection node, and the *intersection* flag for the proof node for $w_3$ is set. Note that proof nodes at the intersection points store no hash value. The second iteration starts from $w_4$, which is the last saved state. New proof nodes for $w_4$, $w_5$ and $c_2$ are added to the proof vector $P$, while $c_3$ is added to the intersection stack $\sqcup_s$. The third iteration starts from $c_3$ and *searchMulti* adds $c_3$ to the proof vector $P$. Note that $w_6$ is added to the intersection stack $\sqcup_s$. In the last iteration, $w_6$ and $c_4$ are added to the proof vector $P$, and nothing is added to the stack. As the intersection stack $\sqcup_s$ is now empty, the loop is over. Note that all proof nodes of the challenged indices have their *end* flags and length values set (lines 9 and 10). When *genMultiProof* returns, the output proof vector should be as in Figure 13.

**Algorithm A.13**: verifyMultiProof Algorithm

**Input**: challenge vector $C$, proof vector $P$, tag vector $T$, local *MetaData*
**Output**: accept or reject

Let $P = (A_0, \ldots, A_k)$, where $A_j = (level_j,\ r_j,\ \text{hash}_j,\ rgtOrDwn_j,\ isInter_j,\ isEnd_j,\ length_j)$ for $j = 0, \ldots, k$, and $T = (tag_0, \ldots, tag_n)$, where $tag_m$ is the tag for the challenged $block_m$ for $m = 0, \ldots, n$.

1 $start = n$
2 $end = n$
3 $t = n$
4 $\text{hash}_{prev} = 0$
5 $startTemp = 0$
6 initialize empty $V, \sqcup_h, \sqcup_i$
// Process each proof node from the end to calculate the hash of the root and the indices of the proven blocks
7 **for** $j = k$ *to* $0$ **do**
8   **if** $isEnd_j$ *and* $isInter_j$ **then**
9     hash = $hash(level_j, r_j, tag_t, \text{hash}_{prev}, length_j)$
10     $t = t - 1$
11     $updateRankSum(length_j, V, start, end)$ // Calculate the index values of proven blocks at the leaf level of the current branch of the proof path
12     $start = start - 1$
13   **else if** $isEnd_j$ **then**
14     **if** $t \neq n$ **then**
15       add $\text{hash}_{prev}$ to $\sqcup_h$
16       add $(start, end)$ to $\sqcup_i$
17       $start = start - 1$
18       $end = start$
19     hash = $hash(level_j, r_j, tag_t, \text{hash}_j, length_j)$
20     $t = t - 1$
21   **else if** $isInter_j$ **then**
22     $(startTemp, end) = \sqcup_i.\text{pop}()$
23     $updateRankSum(r_{prev}, V, startTemp, end)$ // Last stored indices of the proven blocks are updated using the rank state of the current intersection
24     hash = $hash(level_j, r_j, \text{hash}_{prev}, \sqcup_h.pop())$
25   **else if** $rgtOrDwn_j = rgt$ *or* $level_j = 0$ **then**
26     hash = $hash(level_j, r_j, \text{hash}_j, \text{hash}_{prev})$
27     $updateRankSum(r_j - r_{prev}, V, start, end)$ // Update indices of the proven blocks that are on the current branch of the proof path
28   **else**
29     hash = $hash(level_j, r_j, \text{hash}_{prev}, \text{hash}_j)$
30   $\text{hash}_{prev}$ = hash
31   $r_{prev} = r_j$
//*endnodes* is a vector of proof nodes marked as *end*, in the order of appearance in $P$
32 **if** $\forall a,\ 0 \le a \le n,\ 0 \le C_a - V_a < endnodes_{n-a}.length$ *AND* $hash = MetaData$ **then**
33   return accept
34 return reject

The hash for each proof node can be calculated in different ways as described below using the example of Figures 12 and 13. The hash calculation always has the *level* and *rank* values stored in a proof node as its first two arguments.

- If a proof node is marked as *end* but *not intersection* (e.g., $c_4$, $c_2$, and $c_1$), this means the corresponding

node was challenged (to be checked against the challenged indices later), and thus its tag must exist in the tag vector. We compute the corresponding hash value using that tag, the hash value stored in the proof node (null for $c_4$ since it has no $after$ neighbor, the hash value of $v_4$ for $c_2$, and the hash value of $v_3$ for $c_1$), and the corresponding length value (110 for $c_4$, 80 for $c_2$ and $c_1$).

- If a proof node is not marked with either flag and, $rgtOrDwn = rgt$ or $level = 0$ (e.g., $w_6$, $w_2$), this means the $after$ neighbor of the node is included in the proof vector and the hash value of its $below$ is included in the associated proof node (if the node is at the leaf level, the tag is included instead). Therefore we compute the corresponding hash value using the hash value stored in the corresponding proof node and the previously calculated hash value of the $after$ neighbor (hash of $c_4$ is used for $w_6$, hash of $w_3$ is used for $w_2$).
- If a proof node is marked as *intersection and end* (e.g., $c_3$), this means the corresponding node was both challenged (thus its tag must exist in the tag vector) and is on the proof path of another challenged node; therefore, its $after$ neighbor is also included in the proof vector. We compute the corresponding hash value using the corresponding tag from the tag vector and the previously calculated hash value of the $after$ neighbor (hash of $w_6$ for $c_3$).
- If a proof node is marked as *intersection* but *not end* (e.g., $w_5$ and $w_3$), this means the node was not challenged but both its $after$ and $below$ are included in the proof vector. Hence, we compute the corresponding hash value using the previously calculated two hash values of its $after$ and $below$ neighbors (the hash values calculated for $c_2$ and for $c_3$, respectively, are used for computing the hash of $w_5$, and the hash values calculated for $c_1$ and for $w_4$, respectively, are used for $w_3$).
- If none of the above is satisfied, this means a proof node has only $rgtOrDwn = dwn$ (e.g., $w_4$ and $w_1$), meaning the $below$ neighbor of the node is included in the proof vector. Therefore we compute the corresponding hash value using the previously calculated hash value of this $below$ neighbor (hash of $w_5$ is used for $w_4$, and hash of $w_2$ is used for $w_1$) and the hash value stored in the corresponding proof node.

## A.6 Handling Multiple Updates at Once

**Algorithm A.14**: multiUpdate Algorithm

```
  Input: FlexList, U
  Output: P, T, M, newRootHash
  Let U = (u_0, ..., u_k) where u_j is the j^th update information
1 C = generateIndices(U) //According to the nature of the update for each u ∈ U , we add an
  index to the vector (u_j.i for insert and modify, u_j.i and u_j.i − 1 for remove as it is for
  a single update proof)
2 P, T, M= genMultiProof(C) //Generates the multiProof using the FlexList
3 for i = 0 to k do
4     apply u_i to FlexList without any hash calculations
5 update C to all affected nodes using U
6 calculateMultiHash(C) // Calculates hash values of the changed nodes
7 newRootHash = FlexList.root.hash
```

**Algorithm A.15**: calculateMultiHash Algorithm

```
  Input: C
  Output:
  Let C= (i_0, ..., i_k) where i_j is the (j + 1)^th altered index; state_m = (node_m, lastIndex_m, rs_m)
1 cn = root; rs = 0; ⊔_s, ⊔_l are empty; state= (root, k, rs)
  // Call hashMulti method for each index to fill the changed nodes stack ⊔_l
2 for x = 0 to k do
3     hashMulti(state.node, C, x, state.end, state.rs, ⊔_l, ⊔_s)
4     if ⊔_s not empty then
5         state = ⊔_s.pop(); cn = state.node ; state.rs += cn.below.r
6 for k = ⊔_l.size to 0 do
7     calculate hash of k^th node in ⊔_l
```

**Algorithm A.16**: hashMulti Algorithm

```
Input: cn, C, first, last, rs, ⊔_l, ⊔_s
Output: cn, ⊔_l, ⊔_s
  // Index of the challenged block (key) is calculated according to the current sub skip
  list root
1 i = C_first − rs
2 while Until challenged node is included do
3     cn is added to ⊔_l
      //When an intersection is found with another branch of the proof path, it is saved
      to be continued again, this is crucial for the outer loop of ``multi'' algorithms
4     if isIntersection(cn, C, i, last_k, rs) then
          //note that last_k becomes last_{k+1} in isIntersection method
5         state(cn.after, last_{k+1}, rs+cn.below.r) is added to ⊔_s
6     if (CanGoBelow(cn, i)) then
7         cn = cn.below //unless at the leaf level
8     else
          // Set index and rank state values according to how many bytes at leaf nodes are
          passed while following the after link
9         i -= cn.below.r; rs += cn.below.r; cn = cn.after
```

**hashMulti** (Algorithm A.16), employed in *calculateMultiHash* algorithm, collects nodes on a search path of a searched node. It also collects the intersection points (the lowest common ancestor (lca) of the node the collecting is done for and the next node of which the hash calculation is needed). The repetitive calls from *calculateMultiHash* algorithm for each searched node collect all nodes which may need a hash recalculation. Note that each time, a new call starts from the last intersecting (lca) node.

**calculateMultiHash** (Algorithm A.15) first goes through all changed nodes and collects their pointers, then calculates all their hash values from the largest index value to the smallest, until the root. This order of hash calculation respects all hash dependencies.

We illustrate handling multiple updates with an example. Consider a *multiUpdate* called on the FlexList of Figure 3 and a consecutive modify and insert happen to indices 50 and 110 respectively (insert level is 2). When the updates are done without hash calculations, the resulting FlexList looks like the one in Figure 14. Since the tag value of $c_6$ has changed and a new node added between $c_6$ and $c_7$, all the nodes getting affected should have a hash recalculation. If we first perform the insert, we need to calculate hashes of $n_3$, $n_2$, $c_6$, $n_1$, $c_2$ and $c_1$. Later, when we do the modification to $c_6$ we need to recalculate hashes of nodes $c_6$, $n_1$, $c_2$ and $c_1$. There are 6 different nodes to recalculate hashes of, but we do 10 hash calculations. Instead, we propose performing the insert and modify operations and call *calculateMultiHash* to indices 50 and 110, which calculates the necessary hash values without any wasted effort.

Remember that in the example of Figure 14, *calculateMultiHash* was called with indices 50 and 110. The first call of *hashMulti* goes through $c_1$, $c_2$, $n_1$, and $c_6$. On its way, it pushes $n_2$ to a stack since the next iteration of *hashMulti* starts from $n_2$. Then, with the second iteration of *calculateMultiHash*, $n_2$ and $n_3$ are added to the stack. At the end, we call the nodes from the stack one by one and calculate their hash values. Note that the order preserves the hash dependencies.

**Algorithm A.17**: constructTemporaryFlexList Algorithm

**Input**: $P, T$
**Output**: root (temporary FlexList)

Let $P = (A_0, \ldots, A_k)$, where $A_j = (\ level_j,\ r_j,\ \text{hash}_j,\ rgtOrDwn_j,\ isInter_j,\ isEnd_j,\ length_j)$ for $j = 0, \ldots, k$; $T = (tag_0, \ldots, tag_t)$, where $tag_t$ is tag for challenged $block_t$ and dummy nodes are nodes including only hash and rank values set on them and they are final once they are created; //

1 $root$ = new Node($r_0, length_0$) // This node is the root and we keep this as a pointer to return at the end//
2 $\sqcup_s$ = new empty stack
3 $cn$ = $root$
4 dumN = new dummy node is created with hash$_j$
5 $cn$.after = dumN
6 **for** $i = 0$ *to* $k$ **do**
7     $nn$ = new node is created with Level$_{i+1}$ and $r_{i+1}$
8     **if** $isEnd_i$ *and* $isInter_i$ **then**
9         $cn.tag$ = next tag in $T$; $cn.length$ = $length_i$ ; $cn$.after = $nn$; $cn$ = $cn$.after
10     **else if** $isEnd_i$ **then**
11         $cn.tag$ = next tag in $T$; $cn.length$ = $length_i$ ; **if** $r_i$ != $length_i$ **then**
12             dumN = new dummy node is created with hash$_i$ as hash and $r_i$ - $length_i$ as rank
13             $cn$.after = dumN
14         **if** $\sqcup_s$ *is not empty* **then**
15             $cn$ = $\sqcup_s$.pop() ; $cn$.after = $nn$; $cn$ = $cn$.after
16     **else if** $level_i = 0$ **then**
17         $cn.tag$ = hash$_i$ ; $cn.length$ = $r_i$ - $r_{i+1}$ ; $cn$.after = $nn$ ; $cn$ = $cn$.after
18     **else if** $isInter_i$ **then**
19         $cn$ is added to $\sqcup_s$ ; $cn$.below = $nn$; $cn$ = $cn$.below
20     **else if** $rgtOrDwn_i = rgt$ **then**
21         $cn$.after = $nn$
22         dumN = new dummy node is created with hash$_i$ as hash and $r_i$ - $r_{i+1}$ as rank
23         $cn$.below = dumN ; $cn$ = $cn$.after
24     **else**
25         $cn$.below = $nn$
26         dumN = new dummy node is created with hash$_i$ as hash and $r_i$ - $r_{i+1}$ as rank
27         $cn$.after = dumN ; $cn$ = $cn$.below
28 return $root$

**Algorithm A.18**: verifyMultiUpdate Algorithm

**Input**: $P$, $T$, $MetaData$, U, $MetaData_{byServer}$
**Output**: accept or reject

Let U$= (u_0, \ldots, u_k)$ where $u_j$ is the $j^{th}$ update information

1 **if** *!verifyMultiProof(P, T, $MetaData$)* **then**
2     return reject
3 FlexList = buildTemporaryFlexList($P$)
4 **for** $i = 0$ *to* $k$ **do**
5     apply $u_i$ to FlexList without any hash calculations
6 calculate hash values of all nodes in the temporary FlexList. //A recursive call from the root
7 **if** *root.hash* != $MetaData_{byServer}$ **then**
8     return reject
9 return accept